ESD-TR-87-132

SBIR-86-FINAL

Secure Software Verification Tools

COMPUSEC, Inc.
5333 Mission Center Rd., Suite 100
San Diego, CA 92108

10 February 1987

Prepared For

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR STRATEGIC SYSTEMS
HANSCOM AIR FORCE BASE, MASSACHUSETTS

The technical report titled: "Secure Software Verification Tools " has been reviewed and is approved for publication.

John M Molloy 27 Aug 87
JOHN M. MOLLOY, 1Lt, USAF
Compusec Contract Manager

FOR THE COMMANDER

James D Taylor
JAMES D. TAYLOR, Lt Col, USAF
ESD SBIR Manager
Deputy for Development Plans

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SBIR-86-FINAL	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-87-132
6a. NAME OF PERFORMING ORGANIZATION COMPUSEC, Inc. / 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Hq, Electronic Systems Division (SYC-2)
6c. ADDRESS *(City, State, and ZIP Code)* 5333 Mission Center Rd., Suite 100 San Diego, CA 92108	7b. ADDRESS *(City, State, and ZIP Code)* Hanscom AFB Massachusetts 01731-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Deputy for Strategic Systems / 8b. OFFICE SYMBOL *(If applicable)* ESD/ SYC-2	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-86-C-0203
8c. ADDRESS *(City, State, and ZIP Code)* Hanscom AFB Massachusetts, 01731-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Secure Software Verification Tools

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Final Technical	FROM 86-8-10 TO 87-2-10	1987 February 10	162

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Security, Source Code Verification, Backdoor, Trapdoor, Timebomb Elimination

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*
This Phase I effort demonstrated the viability of automated security verification at the implementation level. Source code verification capability would represent a technology advancement that would greatly aid the development of secure software systems. Currently, source code verification is a tedious process consisting of manual code-to-design correspondence checking. If an implementer deviates from the secure design, whether intentionally or negligently, the discrepancy can be detected through manual analysis. This analysis is only as valid as the skill and expertise of the individual conducting the correspondence check. Furthermore, manual analysis is typically performed only on selected code due to its high cost. Therefore, a large secure software system remains vulnerable to insertion of surreptitious code throughout the implementation phase. This problem can be alleviated by an automated extension of the formal verification process that can operate directly on the source code.

(Cont'd on reverse)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT.] ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL John M. Molloy, 1Lt, USAF	22b. TELEPHONE *(Include Area Code)* (617) 271-5053 / 22c. OFFICE SYMBOL ESD/SYC-2

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Cont. of Block 19

Phase I research focused on the evaluation of two different approaches to the automation of source code:

Source-to-SPECIAL (STOS). STOS is an automated translation of source to SPECIAL that will function as a front-end for the Hierarchical Development Methodology (HDM) verification environment.

Source-to-Formulas (STOF). STOF is an automated translation of source directly to formulas. This innovative concept serves to streamline the verification process by eliminating the need for a specification language.

COMPUSEC concludes that STOF is the approach of choice; representing innovative technical progress in the field of software verification.

PREFACE

COMPUSEC's principle objective during the SBIR Phase I effort was to demonstrate the viability of automated security verification at the implementation level. This Final Technical Report is a compilation of the results of the Phase I investigation. Section 1 introduces relevant source code verification criteria. Section 2 presents the Phase I investigation approach. Section 3 describes criteria for evaluating the verifiability of different high level source languages. Section 4 compares the Source-to-SPECIAL (STOS) and Source-to-Formulas (STOF) translation approaches. Section 5 contrasts three different theorem provers. Section 6 contains specifications for an STOS and an STOF Verification Tool. Finally, conclusions and recommendations are presented in Section 7. Two appendices are included with this Final Report: Appendix A contains evaluations for all considered candidate languages; Appendix B takes a short Ada source program through the STOF translation approach.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

INTRODUCTION

1.1 SOFTWARE THREATS

During software development, subversive software may be inserted that threatens the fielded system's mission. This type of software attack includes: Trojan horse, virus, worm, covert channel (including timing and storage channels), time bombs, trap doors, and backdoors. These attacks fall into one of the following threat scenarios:

- System compromise: Covert channels (timing or storage)
- Corruption of system information: Trojan horse, virus, worm
- Delay or denial of system services: Time bomb, virus
- Misappropriation of system resources: trapdoors, backdoors

1.2 SOFTWARE COUNTERMEASURES

Currently, the development of secure applications requires that special countermeasures be used to protect against subversion in the software development environment. Typical countermeasures combine the of use of cleared software developers, blind purchase of commercial off-the-shelf software, and two-person control. These countermeasures address certain threat scenarios, but do not entirely preclude or prevent the existence of threat mechanisms within the software itself.

Assurance that subversive software cannot succeed in the software development environment can only be achieved by applying comprehensive security awareness and analysis throughout a project's life cycle. Verified development environments can be expected to become a requirement for future government software development projects. Methods of analyzing security-relevant aspects of software design must become embedded in the software engineering methodology. Software developers and their managers require early feedback about security design flaws in order to affect design improvements in a timely, cost-effective manner.

1.3 FORMAL VERIFICATION OF SOFTWARE

When software threat mechanisms are characterized according to rules specified by a security policy, formal verification can be applied to assure that software does not violate those rules. Automated formal verification is a rigorous process that provides feedback in the form of failed proofs that point to questionable software modules. Formal verification has been used for two major categories of security-relevant proofs:

- **Information Flow.** Proofs regarding information flow can ascertain whether a program causes an insecure data flow to occur. [1]

- **Proof-of-Correctness.** Proofs regarding program correctness ascertain whether the preconditions of some processing (X) plus the processing of (X) imply the postconditions of (X). [2]

Several factors affect the degree of assurance that can be obtained by using formal verification techniques:

- **Security Policy.** To what extent is the security policy able to capture all security relevant rules?

- **Security Model.** To what extent is the security model able to capture the essence of security policy statements in mathematical formalisms?

- **Language Choice.** To what extent can the specification or implementation language express software functionality in a manner suitable for static analysis?

- **Theorem Prover Choice.** What is the relative strength of the theorem prover to be used? What performance and time costs are associated with its use?

- **Verification Methodology Choice.** Is the verification methodology endorsed by the National Computer Security Center (NCSC)? Or, does the methodology represent innovative technology improvement suitable for endorsement for use on military projects?

1.3.1 Security Policy And Verification

Several military standards specify software security requirements:

- DoD Directive 5200.28 - Security Requirements for ADP Systems
- DoD Directive 5200.28M - ADP Security Manual
- AFSCP 207-1 - System Security Engineering Management
- AFR 205-16 - Automated Data Processing (ADP) Security Policy Procedures and Responsibilities
- AR 380-380 - Automated System Security
- OPNAVINST 5239.1A - ADP Security Program

These documents represent several different styles of characterizing threats and/or the mechanisms required to address those threats.

Another document, DoD 5200.28-STD, "Trusted Computer System Evaluation Criteria," classifies systems into four broad hierarchical divisions of enhanced security protection. These criteria form a basis for the evaluation of the effectiveness of security controls built into software.

1.3.2 Security Models And Verification

Formal verification analysis is performed with respect to a mathematical security model. This model represents the security policy to be enforced on the system being verified. The totality of protection mechanisms within the system is called its trusted computing base (TCB). The TCB can be mapped to mathematical axioms that formalize security policy rules.

Security policy statements can be precise and system-specific, or can be sufficiently broad to relate to whole classes of secure systems. The security policy standards referenced in the previous section relate to all military applications and describe security policy requirements that must be formally expressed in a mathematical security model in order to be suitable for formal verification.

1.3.3 Language Choice And Verification

Languages differ greatly with respect to ease of translation to intermediate specification languages or formulas. They also differ significantly in their suitability for static security analysis. In general, languages that are unambiguous and restrict information flow are more favorable for translation and for verification. This subject is discussed in further detail in Section 3. Appendix A provides the results of the language evaluation conducted for this Phase I effort.

1.4 THEOREM PROVERS

Theorem provers can be evaluated for their proving capabilities, performance, user interaction, and report generation. An ideal theorem prover is one that (1) has extended proving capabilities (i.e. handles induction, propositional logic, and user-defined recursive functions), (2) has a relatively fast performance rate, (3) is user-friendly, and (4) produces clear, well-organized reports. In large systems requiring formal verification, these characteristics are vital to ensure complete, correct, and understandable results.

Formula notation required as input for each theorem prover also differs. For example, the input for Boyer-Moore theorem prover is lambda calculus [3] in prefix format, while other systems use standard predicate calculus in infix format. [4],[5] However, the specific input format chosen makes little difference to the implementation of the theorem prover. Section 5 contains further information on existing theorem provers.

1.5 VERIFICATION METHODOLOGIES

Three methodologies have been supported by the NCSC for use on military contracts:

- Formal Development Methodology (FDM)
- GYPSY
- Hierarchical Development Methodology (HDM).

1.5.1 FDM

FDM consists of a set of tools and languages, as follows:

- The Ina Jo language for writing specification and requirements
- The InaMod language, an extension of the Ina Jo language, for writing assertions about programs
- The Ina Jo processor, for examining specifications and generating logical assertions
- The Interactive Theorem Prover (ITP), for assisting users in proving the logical assertions generated by the Ina Jo processor.
- An ITP post-processor, for generating transcript files of completed proofs. The post-processor eliminates all steps not actually used in a proof, converts the contents of the file from the ITP internal representation, and reformats the text.

FDM provides the ability to map various levels of requirements and design refinement within a system to the next higher level. Its purpose is to prove adherence of a system specification to a set of logical relations, along with proving consistency between different levels of specification. FDM has been used on several significant multi-level secure systems. FDM has been applied to AUTODIN II, the Secure Transacting Processing Experiment (STPE), a Job Stream Separator (JSS), a kernelized IBM VM (KVM), a Computer Operating System/Network Front-End (COS/NFE), and the Secure Release Terminal. [6,IV]

1.5.2 GYPSY

GYPSY is an integrated system of methods, languages, and tools for building formally verified software systems. The GYPSY methodology provides capability for both system specification and implementation within the verification environment. Its purpose is to support proofs about the correctness of system specifications and programs. In addition, An information flow tool is under development. The GYPSY verification environment includes a GYPSY database manager, a parser, an edit interface, a verification condition generator, the Bledsoe theorem prover, a program optimizer, a Bliss translator, and an Ada translator that operate on specifications developed in GYPSY.

GYPSY has been used to verify several experimental applications, including message switching systems, selected components of an air traffic control system, communications protocols, the trusted applications for SCOMP [7], and the monitoring of inter-process communication. [6,II]

1.5.3 HDM

HDM was developed as an aid to design, implementation, and verification of secure software systems. Two dialects of HDM exist and are commonly called "old HDM" and "Enhanced HDM" (E-HDM). COMPUSEC has used old HDM successfully on the Army's Regency Net project and the Navy's AN/GSC-40, ANGSC-40A, and ANGSC-40B Command Post Terminals. HDM is a highly effective method for analyzing systems with respect to information flow. The toolset includes the language SPECIAL, the MLS formula generator for multilevel security analysis, and the Boyer-Moore theorem prover.

Although HDM has been used to formally verify a number of existing secure systems such as SCIACT and SCOMP [7], questions have surfaced concerning its continued endorsement for use on new systems. In September 1986, the HDM toolset was removed from the NCSC's Endorsed Tools List (ETL). [8] This action occurred for two reasons: first, funds are limited for supporting verification toolsets; second, an enhancement to the toolset is currently being developed by SRI International. E-HDM uses the Shostak rather than Boyer-Moore theorem prover and will operate on a substantially different version of SPECIAL. Revised SPECIAL is purported to be able to progressively specify program design to finally arrive at a specification that is a small unimplemented subset of Pascal. HDM is being used to reason about the specifications of SIFT, an experimental operating system for a fault-tolerant computer. To date, NCSC has not decided whether to support either or both HDM and E-HDM on the ETL. [6,V]

1.5.4 Methodology Advantages And Disadvantages

Use of any of these verification methodologies for security theorem proving has advantages and disadvantages. One advantage is that these methodologies already exist and are accredited by the NCSC for use on military projects. However, each toolset contains some intrinsic limitations in functionality or completeness that must be supplemented with additional verification techniques.

FDM has been criticized for a lack of both formal description of the language and lack of formal logic for reasoning about Ina Jo specifications. The Ina Jo Theorem Prover has been described as restrictive. [6,IV]

Although GYPSY supports BLISS or Ada as an implementation language, implementable code can only be reached after exhaustive specification using the GYPSY language. For any project using more conventional program design languages, the intermediate step of translating this into GYPSY input language would be necessary before verification efforts could continue. Also, although GYPSY has been used on several experimental projects, it has not yet developed a significant track record on fielded systems. [6,II]

COMPUSEC has developed several support tools for use with HDM. These include a Bubble-to-SPECIAL (BTOS) tool for translating Data Flow diagrams into HDM's SPECIAL [9] and an Ada-to-SPECIAL (ATOS) translator that translates Ada program design language (PDL) into SPECIAL. These tools have been extremely useful. However, program designers have often expressed a preference for obtaining security feedback about code rather than just design. This capability currently does not exist.

SECTION 2

INVESTIGATION APPROACH

This outline represents the approach that was followed during this Phase I effort to investigate secure source code verification.

I. **Perform Candidate Language Evaluations.** A method was developed for rating a cross-section of 11 different languages on their suitability for static security analysis and ease of translation.

II. **Perform Translation Approach Evaluations.** Two candidate translation approaches were evaluated by attempting translations of several generic language constructs in two different languages.

- **Evaluate Source-to-SPECIAL (STOS) Approach.** STOS is an automated translation of source to SPECIAL that would function as a front-end for the HDM verification environment.

- **Evaluate Source-to-Formulas (STOF) Approach.** STOS is an automated translation of source directly to formulas. This concept streamlines the verification process by eliminating the need for a specification language.

III. **Perform Theorem Prover Evaluations.** The relative capabilities and ease of use of three different theorem provers was considered:

- Boyer-Moore Theorem Prover
- Shostak Theorem Prover
- COMPUSEC Theorem Prover

IV. **Develop Source Code Verification Tool Specifications.** Functional specifications were developed for two tools that, if developed, could implement the two evaluated approaches.

- Source-to-SPECIAL (STOS) Verification Tool
- Source-to-Formulas (STOF) Verification Tool

V. **Form Conclusions and Recommendations.** Based on the accumulated data, the two source code verification approaches were compared and one was recommended as more favorable.

SECTION 3

CANDIDATE LANGUAGE EVALUATIONS

3.1 LANGUAGE CHARACTERISTICS

Characteristics inherent to the syntax or semantics of a language can favorably or unfavorably impact the translation of the language for static verification analysis. This impact applies to analysis of both information flow and correctness. In general, languages that are unambiguous and that clearly show control of information flow represent better candidates for secure software verification. The following six categories of characteristics that impact static verification analysis have been identified.

- Standardization
- Code Structure
- Ambiguity
- Visibility
- Data Representation
- Interfaces

3.1.1 Standardization

Languages that have a well-defined standard are desirable for translation. Rigorous standards clearly specify syntax and semantics of a language no matter which implementation is chosen. Clear specification promotes automated translation of that language for static analysis. Examples of organizations that promote language standards include the DoD (MIL-STD), ANSI, and ISO. In addition, several less formal standards have developed where one author has become the defacto reference for implementations of a particular language.

3.1.2 Code Structure

The following features affect code structure and are important to security verification analysis.

- Process/task use (Modularity)
- Flow control
- Statement evaluation order
- Recursion
- Backtracking

Modularity of process/task simplifies the problem of verification of large software systems by breaking these systems down into meaningfully separate logical (and possibly physical) components. In addition, modularity tends to reduce the amount of component visibility (e.g., variables, static structures, and types), which can effectively scope the verification effort into smaller, more manageable components.

All languages have some mechanisms for flow control. Control structures may be difficult or easy to use. In addition, these structures may be difficult or easy to translate for verification purposes. Information flow resulting from conditional statements such as "if" and "case" must be captured and represented in the source code verification translation process. "Case" statements are useful because they group related control flow into a single construct.

Statement evaluation order must be deterministic and unambiguous in order to be suitable for translation and static analysis. It is important that the order of evaluation of a single statement can be determined in any context in which it is used.

Recursion is an unfavorable language characteristic during verification [10]. Information flow in recursive processes can be difficult to trace, which inhibits static analysis. On the other hand, loops are often used in languages that do not depend upon recursion. Information flow within loops is more easily traced.

Backtracking is a feature of some languages that is useful for functional reasons, but increases the difficulty of performing static analysis on the resulting code. The repeated scans involved with backtracking, when used by a language compiler, may cause an infinite loop condition. This makes it difficult to perform a meaningful static security analysis. [11]

3.1.3 Ambiguity

Scoping, type coercion, and parameter passing can be ambiguous language characteristics. Scoping mechanisms make it easier to resolve names used in the code. Type checking is desirable, but some type coercions can introduce ambiguity into the information flow. If type coercion capability exists in a language, the rules for its application must be unambiguous in order to support static analysis. It is preferable if type coercion is not allowed.

Procedures and functions encapsulate operations into a single reusable block format. They provide a level of abstraction that potentially reduces the ambiguity of code. In addition, procedures and functions decrease the number of necessarily visible variables. Correct use of procedures and functions makes static verification analysis easier.

Languages with scoping rules that restrict the visibility of entities, do not allow coercion, and have clearly defined mechanisms for parameter passing are desirable. Such rules limit information flow possibilities in any section of code. Because all information flow within a system must be checked during formal verification, these characteristics will significantly decrease the difficulty of the static source code verification analysis.

3.1.4 Visibility

The use of hierarchies, data hiding, and data flow constructs impacts the visibility of data within source code. Data typing constructs in a language can cluster individual entities into a single entity type. This level of abstraction facilitates grouping entities with similar uses or contexts. Grouping may then reflect hierarchies within entities. Grouping can also be used to "hide" data, effectively restricting its visibility. Data hiding can be extremely useful in reducing the number of possible information flows. Ultimately, security verification depends on identification of source and destination of all data flows. It is possible for certain language constructs to obscure this information flow.

3.1.5 Data Representation

Abstract data types and data separation play a role in data representation. Arrays, records, and pointers are all vital language features. Although these constructs can be avoided, they prove useful in most applications. Languages written without these features (such as assembly language) may provide the advantage of being compact and fast, but they lose some measure of readability and reliability. Furthermore, the complexity of an application written in higher level code increases unnecessarily if these constructs are not used. Distinct separation of code and data is also desirable. Self-modifying code is to be avoided.

3.1.6 Interfaces

An ideal language has easy access to extensive operating system (OS) services, explicit definition of all interfaces to all components, a rich I/O construct set with direction-specific information, and a comprehensive user environment. These attributes also support static security analysis.

3.2 CANDIDATE LANGUAGE EVALUATION APPROACH

Eleven candidate languages were evaluated. [12,13,14,15,16,17,18,19,20] The following candidate languages were chosen in order to form a representative cross-section of commonly available high-level languages.

- Ada
- BASIC
- C
- Forth
- FORTRAN
- Lisp
- Modula
- Pascal
- PLM
- PROLOG
- SNOBOL

First, a list was compiled of language attributes impacting code verifiability. (See categories of characteristics described in Section 3.1). These language characteristics and attributes determine if or how well code can be translated into either SPECIAL or formulas. Each language was analyzed for the presence of each attribute. If present, its style of use was then analyzed for its impact on static verification analysis. Analysis resulted in the assignment of a value between 0 and 3 for each attribute, where 3 indicates the most suitable use of the characteristic for static verification analysis. This unweighted criteria rating scheme is provided in Appendix A.1. Results of using this rating scheme are provided in Appendix A.2.

Table 3-1. Summary of Language Evaluation Results

CRITERIA	ADA	MODULA	PASCAL	PLM	C	LISP	PROLOG	FORTRAN	FORTH	SNOBOL	BASIC
TOTALS	270	242	222	214	198	193	165	163	154	151	139
STANDARDIZATION (10)	15	3	4	3	2	1	1	3	1	1	1
CODE STRUCTURE (20)	57	51	41	48	43	30	49	33	48	26	28
AMBIGUITY (20)	60	60	56	52	50	46	46	40	28	54	46
VISIBILITY (15)	45	45	41	41	36	39	25	36	25	30	15
DATA REPRESENTA-TION (15)	40	40	40	25	30	35	25	20	35	20	15
INTERFACES (20)	53	43	40	45	37	42	19	30	17	20	34

In addition to considering suitability for static verification analysis, the importance of each language attribute in the translation process was also evaluated. This was done by dividing 100 points among the six identified criteria categories, and then correlating each allotment to individual attributes within the criteria categories. This system reflects the relative positive impact of each attribute on the translation process. A higher weighting indicates higher importance. Results of application of this weighted rating can be found in Appendix A.3. A summary of language evaluation results is listed in Table 3-1 and in Appendix A.4.

Results of this language investigation point to Ada as the language most suited for code verification. Ada ranked highest in all of the criteria categories used. Ada is governed by a rigorous standard found in MIL-STD-1815A. [12] It is a highly structured language with constructs limiting ambiguity. Strong visibility and data representation rules exist in the language. Ada's weakest characteristics for use in security translation include data separation, I/O constructs, and user environments. However, Ada still ranks average or above average in these categories.

SECTION 4

TRANSLATION APPROACH EVALUATIONS

4.1 STOS APPROACH

This chapter delineates a set of rules for translation of C and Ada source code to SPECIAL. This translation is a reduction of the source code to a high level formal description of source information flow. The rules might need to be tailored to applications where specification conditions exist that alter prcof conditions. For example, in applications that have known properties such as hardware security features, these specifications may not accurately reflect true system characteristics. However, the intention here is to provide general translation rules for applications operating in generic environments.

SPECIAL is a flexible language, intended as a specification for a wide range of applications and programming languages. In the following translation description, specific type, parameter, and assertion sections have been defined. The SPECIAL produced is intended to capture information flow, and does not represent functionality (although SPECIAL is capable of doing so).

The following constructs are described to aid in the understanding of the resulting translations. A number of these terms are inherent to the SPECIAL language (e.g., OFUN, VFUN, NEWVALUE, MODULE). Also described are COMPUSEC-defined parameters (e.g. COND, FUNCTION, CONSTANTS), which aid in the use of SPECIAL as an MLS information flow analysis language.

- **SPECIAL Constructs**

  - **OFUN.** Represents typeless procedures - may be given a subject label.

  - **VFUN.** Represents variables (i.e. integers, characters, arrays, etc.) - may be given an object label.

  - **NEWVALUE.** NEWVALUE is the SPECIAL equivalent of an assignment. It represents flow of information from an expression to either an object or variable. The symbol "'" means NEWVALUE, the symbol FN depicts the number of entities in the function.

Example:

```
'<data> = FN(<data>, <data>, ...  <data>)
```

- COMPUSEC Defined Parameters

  - **COND.** This is the equivalent of a branch conditional. It shows information flow from variables in the conditional expression to the block of statements under the conditionals flow control.

    Example:

    ```
    COND1(<data>), COND2(<data>, <data>), ...    CONDN(<data>,
    <data>, ..<data>)
    ```

  - **FUNCTION.** This is the equivalent of a generic function that is dependent upon input data. It appears on the right hand side (rhs) of expressions, and represents flow of information from every input to the left hand side (lhs).

    Example:

    ```
    F1(<data>), F2(<data>, <data>), ... FN(<data>, <data>, ...
    <data>)
    ```

  - **CONSTANTS.** These represent placeholders and are intended to show correspondence between SOURCE and SPECIAL. LLLLL is a generic label placeholder that is used specifically for a data item's security label information.

    Example:

    ```
    K!
    true
    false
    LLLLL
    ```

4.1.1 Translation Correspondence

When translating a source language to formulas or to a specification language, continuity must be maintained between the resulting translation and the functionality represented in the source code. If the translation accurately represents the functionality of a given source, an observer can easily ascertain the correspondence between source and translation by contextual clues. Also, if the resulting translation maintains the same identifier names as the source, an observer can easily find the correspondence between the source and the translation.

However, practical limitations exist in using functionality to represent correspondence. Consider the representation of functionality in formulas. This representation would include extensive comments and formatting that are not needed in the proof process. The resulting translation would be weighted with extra text that must be processed by tools and stored in machine memory. Hence, much memory space and CPU time would be wasted.

Naming conventions are a more useful method of maintaining correspondence during the translation process. They provide a means of isolating the range of possible source that a given translation represents. This range could then be examined to discover with certainty the exact correspondence. This could be accomplished without adding excessive overhead (i.e., space, CPU time) to the translation process.

The following rules describe the naming convention that will be followed in all subsequent translations. This method uses names to provide contextual clues that indicate source-to-translation correspondence.

- Each identifier contains the name of its declaring block.
- The "#" symbol separates subparts of a name.
- An increasing integer extension is appended to each instance of a data item to disambiguate it from other instances.

Translation of a source identifier to translation identifier will include the name of the declaring block. These entities are readily ascertainable in ALGOL-based languages like Ada and C. An extension describing the scope of each declared item will be created in the form of an increasing integer. In the translation, data item names will be written with their extension to remove any ambiguities (i.e., DATA0, DATA1, .. DATAn). Extensions involving subparts of a data name will also result from concatenating the current extension with each new subprogram body name as entered. When a subprogram body is exited, the corresponding extension is removed. Sharp signs separate subparts of a name, exactly as the dot character in Ada or C. Following this convention, if the subprogram or variable has been declared at the outermost scoping level, no changes to its name will be made.

For example, the extension for a function "B" declared in an unnested subprogram "A" would be "A#B". If variable "C" is declared within function "B," it will be represented as "A#B#C" (this conforms to the Ada naming convention).

4.1.2 Assignment Statements

- **Expression to Scalar Type**

  Statement

```
         Ada              |          C
--------------------------|-----------------------------
x := 42 * y - (z / w);    | x = 42 * y - (z / w)
```

  Translation Rule

  Information flows between every term on the rhs of an assignment statement to every term on the lhs of that assignment.

  Translation Results

```
'x(LLLLL) = F4(K!,y(LLLLL),z(LLLLL),w(LLLLL));
```

- **Structured Type to Same Structured Type**

  Statement

```
         Ada              |          C
--------------------------|-----------------------------
type smallrec is          | typedef struct smallrec {
 record                   |   int   x;
   x : integer;           |   char  y;
   y : character;         | }
 end record;              |
                          |
type stuffrec is          | typedef struct stuffrec {
 record                   |   struct smallrec
  otherstuff : smallrec;  |           otherstuff;
  a,b        : integer;   |   int  a,b;
 end record;              | }
                          |
rec1,rec2 : stuffrec      | struct stuffrec rec1,rec2;
      ...                 |           ...
rec1 := rec2;             | rec1 = rec2;
```

  Translation Rule

  The assignment of a variable of structured type to another variable of the same structured type will produce as many NEWVALUE statements as there are scalar types making up the structured type.

Translation Results

```
'rec1#otherstuff#x(LLLLL) = F1(rec2#otherstuff#x(LLLLL));
'rec1#otherstuff#y(LLLLL) = F1(rec2#otherstuff#y(LLLLL));
'rec1#a(LLLLL) = F1(rec2#a(LLLLL));
'rec1#b(LLLLL) = F1(rec2#b(LLLLL));
```

- **Array Components**

Statement

```
        Ada                |          C
---------------------------|------------------------------
type stuffrec is           |  typedef struct stuffrec {
 record                    |    int   a,b;
   a,b : integer;          |  } stuffarray[10];
 end record;               |
                           |
type twinarray is          |  typedef struct twinarray {
 record                    |    int   x[10],y[10];
  x,                       |  }
  y: array (1..10) of      |
               integer;    |
 end record;               |
                           |
type stuffarray is         |  struct stuffarray rec1;
 array (1..20) of          |  struct twinarray rec2,rec3;
              stuffrec;    |
 rec1       : stuffarray;  |
 rec2,rec3  : twinarray;   |
           ...             |          ...
rec1(u).a := 13 *          | rec1[u].a = 13 *
   rec2.x(v) - rec3.y(w);  |    rec2.x[v] - rec3.y[w];
```

Translation Rule

If a variable of structured type has array type components, then any identifiers that are used to index the array are extracted. Information flow is shown from the array indices on either side of the assignment statement to the variable on the lhs of the assignment.

Translation Results

```
'rec1#a(LLLLL) = F6(u(LLLLL),K!,rec2#x(LLLLL),v(LLLLL),
             rec3#y(LLLLL),w(LLLLL));
```

Discussion

Arrays, records, and pointers are different ways of representing groups of data items. They present a problem in translation because these constructs rarely map to a single unit entity, and are often composed of many multi-component entities. Information flow from using these constructs is non-trivial, and often depends on dynamically changing information. In order for SPECIAL to represent these constructs, the notion of offsets or array indices must be used. The index value used therefore becomes part of the potential information flow in any array operation.

RULE: Information flow from the index always flows to the lhs whether or not the index exists on the lhs or rhs.

Records must be handled similarly, where lowest level components represent offsets indexed into a data structure. However, unlike array indices, records are usually static.

RULE: Break records into component level information. No information is really stored at the top level. This is a structural device. Actual variables can be seen as components within this structure.

Pointers combine aspects of both arrays and record structures. There is no clear rule for handling pointers. A pointer could possibly be shown as a base plus offset.

4.1.3 Conditionals and Iteration

- **"If" Statements**

Statement

```
        Ada                 |          C
----------------------------|-------------------------------
if y > z * aray(i) then     | if (y > (z * aray[i]))
  ii := jj;                 |    {
else                        |      ii = jj;
  kk := ll;                 |    } else {
  vv := ww;                 |      kk = ll;
end if;                     |     vv = ww;
                            |    }
```

Translation Rule

Information flows from all identifiers appearing in the condition part of the "if" or "else if" statement to all outputs of any statements appearing in the "then" or "else" part of the "if" statement. All identifiers appearing in the condition are collected, an implication statement is generated with these identifiers placed on the lhs, and all statements within the scope of the "if" are placed on the rhs.

Translation Results

```
(COND4(y(LLLLL),z(LLLLL),aray(LLLLL),i(LLLLL)) =>
  ( 'ii(LLLLL) = F1(jj(LLLLL)) AND
    'kk(LLLLL) = F1(ll(LLLLL)) AND
    'vv(LLLLL) = F1(ww(LLLLL))
));
```

Discussion

SPECIAL has no inherent limitations for representing conditionals. In general, easier translations result if runtime processing of conditionals is deterministic. A language with a known and static evaluation process at runtime will be more easily translated than one with an unknown or dynamic runtime evaluation process. As an example of non-deterministic processing, consider the following C conditional:

```
if ((x = y) && (k == z)) { ... }
```

In this example, information flows from y to x, and from y and x to the statements within the conditionals block. Information does not necessarily flow from k and z to the statements within the conditionals block because this part of the expression is only evaluated if "(x = y)" is true. Because C short-circuits conditional expression evaluation, actual information flow may only be known at runtime. Information therefore becomes dependent upon partial evaluation of complex expressions. Dynamic or unknown evaluation processes of conditional expressions may cause some inaccuracy in the representation of information flow. These inaccuracies are cosmetic in the sense that they are not faults but merely fail to show the true nature of the information flow possibilities.

Clearly, a language with static conditional expression evaluation makes for a translation with more accurate representation of information flow. In this case, the easiest solution is to ignore the problem. Short-circuiting conditional expression evaluation can only lessen the total information flow to statements within the conditional block. Therefore, no flow will be lost by not representing short-circuiting.

• "Case" Statements

Statement

```
        Ada                    |          C
-------------------------------|------------------------------
  case (x) of                  |  switch (x) {
    when 1 =>                  |    case 1:
      i := j;                  |      i = j;
      m := k;                  |      m = k;
                               |      break;
      others =>                |    default:
      c := n;                  |      c = n;
                               |      break;
  end case;                    |  }
```

Translation Rule

The "case" construct in ALGOL-based languages represents, as a translation problem, a special case of a sequence of "if" conditionals. It may be represented in the same manner as a series of "if" statements, with the same conditional expression.

Translation Results

```
    (COND1(x(LLLLL)) =>
      ( 'i(LLLLL) = F1(j(LLLLL)) AND
       'k(LLLLL) = F1(l(LLLLL)) AND
       'm(LLLLL) = F1(n(LLLLL))
     ))
```

• Conditional Loops

Statement

```
        Ada                    |          C
-------------------------------|------------------------------
 stuff:                        |  stuff:
   loop                        |  do {
     x := x + 1;               |    x += 1;
     exit when x > 42;         |    if (x > 42) break;
   end loop stuff;             |  } while (1 == 1);
```

Translation Rule

If a name for the loop exists, it will be extracted. If there are no conditions for the loop (however, there may be an "if" statement that leads into an unconditional exit), then the statements within the loop will be processed as if the loop did not exist.

Translation Results

```
(COND2(x(LLLLL),K!) => ('x(LLLLL) = F2(x(LLLLL),K!) ));
```

- **Iterative Loops**

Statement

```
        Ada                |          C
---------------------------|-------------------------------
 for a in b..42            | for (a = b; a <= 42; a++)
   loop                    |   y *= a;
     y := a * y;           |
   end loop;               |
```

Translation Rule

Any identifiers appearing in the iteration scheme of a "for" statement, in the conditional part of a "while" statement, or as conditions for loop exit will be collected and put on the lhs of an implication, with the rhs being the translation of any statements within the loop.

Translation Results

```
   (COND3(a(LLLLL),b(LLLLL),K!) =>
     ( 'y(LLLLL) = F2(a(LLLLL),y(LLLLL)));
```

Discussion

Loops are not a natural construct in SPECIAL. Loops in SPECIAL are meaningless, because SPECIAL is intended for static analysis. In addition, loops may present problems in translation because they contain potential data flow from control variables to all data items being processed within the loop. Control variables cannot simply be defined as the loop counter in a "for" statement, or the Boolean in a "while" conditional. Any variable that can cause termination or affect the primary control variables can be considered to be able to control the loop. In some cases, loops may be best represented as conditional statements.

In some languages, such as Ada and C, loop execution may be altered using "break," "continue," or other instructions. These instructions may in turn depend upon conditionals that are not apparently part of the loop control variables.

All these dependencies must be identified and then used in the translation to SPECIAL.

RULE: Capture and identify all loop control mechanisms.

RULE: Statements capable of altering the performance of loops must be identified, and their dependencies translated into SPECIAL.

4.1.4 Parameters and Function "RETURN" Statements

- **Parameters**

Statement

```
          Ada               |            C
----------------------------|------------------------------
Procedure A(                |  void A(a, b, c)
  a : in integer;           |
  b : out integer;          |  int   a;
  c : in out integer) is    |  int   *b;
  begin                     |  int   *c;
    b := a + c;             |  {
    c := a * c;             |    *b = a + (*c);
  end;                      |    *c = a * (*c);
                            |  }
Procedure B is              |void B()
  x,y,z : integer;          |  int  x, y, z;
  begin                     |  {
    x := ...;               |    x = ...;
    z := ...;               |    z = ...;
    A(x,y,z);               |    A(x,&y,&z);
  end;                      |  }
```

Translation Rule

- "In" parameters - prior to the EFFECTS_OF statement, a NEWVALUE statement should be generated to show flow from the actual parameter to the new name generated for formal parameter (see Section 4.1.1 for naming conventions). All C parameters act as "in" parameters.

- "Out" parameters - following the EFFECTS_OF statement, a NEWVALUE statement should be generated to show flow from the the formal parameter (see Section 5.1 for discussion on naming conventions for parameters) to the actual parameter.

- "In out" parameters - produce the results of both "in" and "out" parameters.

Translation Results

```
OFUN A [LEVEL L]
EFFECTS
'A#b(LLLLL) = F2(A#a(LLLLL),A#c(LLLLL));
'A#c(LLLLL) = F2(A#a(LLLLL),A#c(LLLLL));

OFUN B[LEVEL L]
EFFECTS
'B#x(LLLLL) = ...;
'B#z(LLLLL) = ...;
'A#a(LLLLL) = F1(B#x(LLLLL));
'A#c(LLLLL) = F1(B#z(LLLLL));
EFFECTS_OF A;
'B#y(LLLLL) = F1(A#b(LLLLL));
'B#z(LLLLL) = F1(A#c(LLLLL));
```

Discussion

In all NEWVALUE statements generated, if the parameters have a record type, then the assignments must be expanded into assignments between components of the actual and formal parameters. (This is the same as the processing of assignment statements involving record variables, see assignment section.)

If there is an expression appearing as the actual of an "in" parameter, then an Fn(...) could be generated to keep the numbers and placement of actual and formal parameters consistent, i.e.

```
A(x*y,z);
```

Becomes

```
EFFECTS_OF A(F2(x(LLLLL),y(LLLLL)),z(LLLLL));
```

- **Function "RETURN" Statements**

Statement

```
        Ada                   |          C
------------------------------|-----------------------------
function A(                   |  int A(x, y)
   x : in integer;            |  int x;
  y : out integer) return     |  int *y;
              integer is      |
  begin                       |
    y := x * 2;               |  {
    return y - 1;             |    *y = x * 2;
  end;                        |    return (*y - 1);
                              |  }
procedure B is                |  void B()
  begin                       |  {
    a := 1;                   |    a = 1;
    c := A(a,b) - 42;         |    c = A(a,&b) - 42;
  end                         |  }
```

Translation Rule

If the subprogram returns a value, then a new name for the returned value could be made by concatenating the subprogram name with returns. This new name should be used in any place that the function value is used. An EFFECTS_OF statement should be placed preceding the use of the new name.

Translation Results

```
OFUN A [LEVEL L]
EFFECTS
'A#y(LLLLL) = F2(A#x,k!);
'A#returns(LLLLL) = F2(A#y(LLLLL),k!);

OFUN B[LEVEL L]
EFFECTS
'a(LLLLL) = F1(k!);
'A#x(LLLLL) = F1(a(LLLLL));
EFFECTS_OF A;
'b(LLLLL) = F1(A#y(LLLLL));
'c(LLLLL) = F2(A#returns(LLLLL),k!);
```

4.2 STOF APPROACH

STOF merges two steps of the verification process into one. The translation of source to formal top-level specification (FTLS) followed by reduction of FTLS to a set of verification formulas is replaced by the single step of generation of formulas from the source.

The simplest form of a formula describing a security condition has the form "lteq(X(label1),Y(label2))." Such a formula will be true if the security label of X, label1, is less than or equal to the label of Y, label2. A potential security violation exists if the label of Y is less than the label of X. The term lteq is used rather than "<=", because the ordering of labels may not be a linear ordering (security labels will always be partially ordered).

One lteq formula is generated for each pair of data items between which information flows. A simple example of an information flow is the assignment statement:

```
Y := X;
```

There is an information flow from X to Y. This is an example of direct information flow. In this case, the security label of X must be less than or equal the security label of Y. Another simple example is the "if" statement:

```
if x < 10 then
    z := 1
else
    z := 2
end if;
```

In the "if" statement above, there is an indirect information flow from x to z, and a formula will be generated comparing the label of x to the label of z. The existence of the flow can be seen by noting that examination of the value of z will determine a range of values for x. A combination of several inferences such as this one might be combined to precisely determine the value of x. If x were labelled secret and z were labelled unclassified then secret information could potentially be inferred from unclassified information.

Throughout the discussion of STOF, the formula "lteq(x,y)" will be taken to mean "lteq(x(label1),y(label2))." It will be assumed that there is a database, such as a data dictionary, which contains all given security label information. This database can be accessed by STOF and searched to determine the label of any data item appearing within an lteq formula.

The rules for determining information flow within a source program are independent of verification--they are part of the semantics of the source language. To generate formulas directly from source code, the rules about information flow within the source language are implemented as a translation program. Two examples of general rules are:

- Information flows from the rhs of an assignment statement to the lhs of the assignment.

- Information flows from any data items appearing in the conditional part of an "if" statement to any output of the statements appearing within the branches of the "if" statement.

The amount of detail showing which outputs are dependent on which inputs determines how specific the verification formulas will be. If an lteq formula is generated for every combination of input and output of a procedure, then it is certain that all information flow has been captured. However, some of the formulas may represent information flow that does not exist. The processing must be examined to prevent generation of spurious formulas.

The generation of formulas for all combinations of input and output may at times be appropriate. During development some modules may be completed before others, and testing of the system in an unfinished state is possible using the interfaces for the uncompleted portions of the system and assuming the worst case (of information flow) for those portions.

The following formulas are represented in both a prefix and infix notation. The semantic meanings of these notations are identical, only the syntactics differ. Prefix notation places the operators at the beginning of the corresponding operands, while infix notation places them in the middle of the operands.

4.2.1 Assignment Statements

- **Expression to Scalar Type**

  Statement

```
        Ada                 |          C
----------------------------|------------------------------
  x1 := 42 * y1 -           |  x1 = 42 * y1 -
            (z2 / x2);      |           (z2 / x2);
```

  Translation Rule

  Information flows from all data items on the rhs of an assignment statement to the data item on the lhs of the assignment statement. The following flows are present in this example:

  42 -> x1, y1 -> x1, z2 -> x1, x2 -> x1

  Resulting Formulas

  - Flow from constants is captured - constants may or many not be given security labels (same in Infix standard as in Prefix Boyer-Moore)

    lteq(42,x1), lteq(y1,x1), lteq(z2,x1), lteq(x2,x1)

- Flow from constants is ignored (same in Infix standard as in Prefix Boyer-Moore)

```
lteq(y1,x1),  lteq(z2,x1),  lteq(x2,x1)
```

- **Structured Type to Same Structure Type**

Statement

```
        Ada                   |         C
------------------------------|-----------------------------
type smallrec is              |  typedef struct smallrec {
 record                       |    int   x;
   x : integer;               |    char  y;
   y : character;             |  }
 end record;                  |
                              |
type stuffrec is              |  typedef struct stuffrec {
 record                       |  struct smallrec otherstuff;
  otherstuff : smallrec;      |    int  a,b;
  a,b        : integer;       |  }
 end record;                  |
                              |
rec1,rec2 : stuffrec          | struct stuffrec rec1,rec2;
      ...                     |             ...
rec1 := rec2;                 |  rec1 = rec2;
```

Translation Rule

Information flows between a component of the record variable on the rhs of the assignment to the corresponding component of the record variable on the lhs of the assignment.

The following flows are present in the above example:

```
rec2#otherstuff#x -> rec1#otherstuff#x,
rec2#otherstuff#x -> rec1#otherstuff#x,
rec2#a            -> rec1#a,
rec2#b            -> rec1#b
```

Resulting Formulas

(Same in Infix standard as in Prefix Boyer-Moore.)

```
lteq(rec2#otherstuff#x,rec1#otherstuff#x),
lteq(rec2#otherstuff#x,rec1#otherstuff#x),
lteq(rec2#a,rec1#a),
lteq(rec2#b,rec1#b)
```

* Array Components

Statement

```
        Ada                  |         C
-----------------------------|-----------------------------
type stuffrec is             |  typedef struct stuffrec {
 record                      |    int   a,b;
   a,b : integer;            |  } stuffarray[10];
 end record;                 |
                             |
type twinarray is            |  typedef struct twinarray {
 record                      |    int   x[10],y[10];
  x,                         |  }
  y: array (1..10) of        |
              integer;       |
 end record;                 |
                             |
type stuffarray is           |  struct stuffarray  rec1;
 array (1..20) of            |  struct twinarray rec2,rec3;
             stuffrec;       |
                             |
 rec1      : stuffarray;     |
 rec2,rec3 : twinarray;      |
           ...               |           ...
rec1(u).a :=  13 *           |  rec1[u].a = 13 *
   rec2.x(v) - rec3.y(w);    |     rec2.x[v] - rec3.y[w];
```

Translation Rule

If a variable of structured type has array-type components, then any identifiers used to index the array are extracted. Information flow is shown from the array indices on either side of the assignment statement to the variable on the lhs of the assignment.

- Information flows from the array variable on the rhs of the assignment statement to the identifier on the lhs.

- Information flows from any identifiers used to index the array variable on the rhs to the identifier on the lhs.

- Information flows from any identifiers used to index an array variable appearing on the lhs to the identifier on the lhs.

The following flows are present in the above example:

```
13 -> rec1#a,    rec2#x -> rec1#a,    rec3#y -> rec1#a
u  -> rec1#a,    v      -> rec1#a,    w      -> rec1#a
```

Resulting Formulas

(Same in Infix standard as in Prefix Boyer-Moore)

```
lteq(13,rec1#a), lteq(rec2#x,rec1#a), lteq(rec3#y,rec1#a)
lteq(u,rec1#a),  lteq(v,rec1#a),      lteq(w,rec1#a)
```

4.2.2 Conditionals And Iteration

- "If" Statements

Statement

```
        Ada               |         C
--------------------------|------------------------------
 if (x1 < y1) then        | if (x1 < y1)
   z1 := x2               |   z1 = x2;
     else                 |     else
       z2 := y2           |   z2 = y2;
 end if;                  |
```

Translation Rule

There is information flow from any data items involved in the conditional part of an "if" statement to any outputs of any statements contained within the "if" statement.

The following flows are present in this example:

```
x1 -> z1,       x1 -> z2,       y1 -> z1,
y1 -> z2,       x2 -> z1,       y2 -> z2
```

Resulting Formulas

- Omitting conditional information (same in Infix standard as in Prefix Boyer-Moore)

```
lteq(x1,z1),   lteq(x1,z2),  lteq(y1,z1)
lteq(y1,z2),   lteq(x2,z1),  lteq(y2,z2)
```

- Retaining conditional information

Infix (standard)	Prefix (Boyer-Moore)
(x1 < y1) => lteq(x1,z1)	(IMPLIES (LESSP x1 y1) (lteq x1 z1))
not (x1 < y1) => lteq(x1,z2)	(IMPLIES (NOT (LESSP x1 y1)) (x1 z2))
(x1 < y1) => lteq(y1,z1)	(IMPLIES (LESSP x1 y1) (lteq y1 z1))
not (x1 < y1) => lteq(y1,z2)	(IMPLIES (NOT (LESSP x1 y1)) (lteq y1 z2))
(x1 < y1) => lteq(x2,z1)	(IMPLIES (LESSP x1 y1) (lteq x1 z1))
not (x1 < y1) => lteq(y2,z1)	(IMPLIES (NOT (LESSP x1 y1)) (lteq y2 z1))

- **"Case" Statements**

Statement

```
          Ada                  |          C
-------------------------------|--------------------------------
  case x1 is                   |  switch (x1) {
    when 1  => x2 := z1;       |      case 1 : x2 = z1;
    when 2  => y2 := y1;       |      case 2 : y2 = y1;
    when 3  => z2 := x1;       |      case 3 : z2 = x1;
    when others =>             |      default:
                x2 := z2;      |               x2 = z2;
  end case;                    |  }
```

Translation Rule

Information flows from the expression part of the "case" to any output of a statement within each alternative of the "case". In both Ada and C the values used to denote each alternative are constants (string or enumerated) and for this reason can usually be ignored.

The following flows are present in the above example:

```
x1 -> x2,     x1 -> y2,     x1 -> z2,     x1 -> x2,
z1 -> x2,     y1 -> y2,     x1 -> z2,     z2 -> x2,
```

(The flows (x1 -> z2) and (x1 -> x2) appear twice, one of each of these can be omitted during the proof process, with the single remaining formula of each representing both possibilities of each.)

<u>Resulting Formulas</u>

- Without using conditional or value information about 'x2' (same in Infix standard as in Prefix Boyer-Moore)

  lteq(x1,x2), lteq(x1,y2), lteq(x1,z2), lteq(x1,z2),
  lteq(z1,x2), lteq(y1,y2), lteq(z2,x2)

- Using conditional but not value information about "x2"

Infix (standard)	Prefix (Boyer-Moore)
cond1(x1) => lteq(x1,x2)	(IMPLIES (COND1 x1) (lteq x1 x2))
cond1(x1) => lteq(z1,x2)	(IMPLIES (COND1 x1) (lteq z1 x2))
cond1(x1) => lteq(x1,y2)	(IMPLIES (COND1 x1) (lteq x1 y2))
cond1(x1) => lteq(y1,y2)	(IMPLIES (COND1 x1) (lteq y1 y2))
cond1(x1) => lteq(x1,z2)	(IMPLIES (COND1 x1) (lteq x1 z2))
cond1(x1) => lteq(x1,x2)	(IMPLIES (COND1 x1) (lteq x1 x2))
cond1(x1) => lteq(z2,x2)	(IMPLIES (COND1 x1) (lteq z2 x2))

- Using conditional and value information about "x2"

```
       Infix (standard)     |    Prefix (Boyer-Moore)
----------------------------|---------------------------
(x1 = 1) => lteq(x1,x2)|(IMPLIES (EQUAL x1 1)
                            |             (lteq x1 x2))
                            |
(x1 = 1) => lteq(z1,x2)|(IMPLIES (EQUAL x1 1)
                            |             (lteq z1 x2))
                            |
(x1 = 2) => lteq(x1,y2)|(IMPLIES (EQUAL x1 2)
                            |             (lteq x1 y2))
                            |
(x1 = 2) => lteq(y1,y2)|(IMPLIES (EQUAL x1 2)
                            |             (lteq y1 y2))
                            |
((x1 = 3) or (x1 = 4))      |(IMPLIES (OR
          => lteq(x1,z2)|           (EQUAL x1 3)
                            |           (EQUAL x1 4))
                            |         (lteq x1 z2))
                            |
(x1 <> 1) and (x1 <> 2)|
and (x1 <> 3)               |
and (x1 <> 4) =>            |(IMPLIES (AND
lteq(x1,x2)                 |           (NOT (EQUAL x1 1))
                            |           (NOT (EQUAL x1 2))
                            |           (NOT (EQUAL x1 3))
                            |           (NOT (EQUAL x1 4)))
                            |         (lteq x1 x2))
                            |
(x1 <> 1) and (x1 <> 2)|
and (x1 <> 3)               |
and (x1 <> 4) =>            |(IMPLIES (AND
lteq(z2,x2)                 |           (NOT (EQUAL x1 1))
                            |           (NOT (EQUAL x1 2))
                            |           (NOT (EQUAL x1 3))
                            |           (NOT (EQUAL x1 4)))
                            |         (lteq z2 x2))
```

- **Conditional Loops**

Statement

```
         Ada              |          C
--------------------------|-----------------------------
 stuff:                   | stuff:
    loop                  |    do {
      x1 := x1 + y1;      |    x1 = x1 + y1;
      exit when x2 > 42;  |    if (x2 > 42) break;
    end loop stuff;       |    } while  (1 == 1);
```

Translation Rule

If a name for the loop exists, then it will be extracted. If there are no conditions for the loop (there may be an "if" statement that leads into an unconditional exit, however), then the statements within the loop will be processed as if the loop did not exist.

The following flows are present in the above example:

```
x2 -> x1,     42 -> x1,     x1 -> x1,     y1 -> x1
```

Resulting Formulas

(Same in Infix standard as in Prefix Boyer-Moore)

```
lteq(x2,x1), lteq(42,x1), lteq(x1,x1), lteq(y1,x1)
```

- **Iterative Loops**

Statement

```
        Ada               |          C
--------------------------|------------------------------
  for x1 in y1..42        |  for (x1 = y1;
    loop                  |       x1 <= 42;
      y2 := x1 * z1;      |       x1++) y2 = x1 * z1;
  end loop;               |
```

Translation Rule

Any identifiers appearing in the iteration scheme of a "for" statement, or in the conditional part of a "while" statement, or as conditions for exiting the loop will be collected, and put on the lhs of an implication, with the rhs being the translation of any statements within the loop.

The following flows are present in the above example:

```
x1 -> y2,     y1 -> y2,     42 -> y2,
x1 -> y2,     z1 -> y2
```

Resulting Formulas

(Same in Infix standard as in Prefix Boyer-Moore)

```
lteq(x1,x2),  lteq(y1,y2),  lteq(42,y2),
lteq(x1,y2),  lteq(z1,y2),
```

4.2.3 Parameters and Function "RETURN" Statements

- **Parameters and Function "Return"**

Statement

```
        Ada                       |          C
--------------------------|-----------------------------
 function A(               |  A(x1,y1)
       x1,y1 : integer)   |  int x1,y1;
     return integer is    |  {
   y2 :integer;           |    int y2;
   begin                  |    y2 = x1 * y1;
    y2 := x1 * y1;        |    return (y2-1);
    return y2 - 1;        |  }
   end;                   |
                          |
 procedure B is           |  B()
   x1,y1,z1 : integer;    |  {
   begin                  |    int x1,y1,z1;
     x1 := 1;             |      x1 = 1;
     y1 := 2;             |      y1 = 2;
     z1 := A(x1,y1) - 42;|      z1 = A(x1,y1) - 42;
   end                    |  }
```

Translation Rule

- Subprogram parameters:

  "In" parameters - there is information flow from the actual parameter to the formal parameter. All C parameters act as "in" parameters.

  "Out" parameters - there is information flow from the formal parameter to the actual parameter.

  "In out" parameters - there is bidirectional flow between the formal parameter and the actual parameter.

- Return statements:

  There is information flow from any data items appearing in a return statement to any expression in which the function is invoked.

  Renaming conventions are as described earlier.

The direct flows present:

```
1    -> B#x1,        2         -> B#y1,
B#y1 -> A#y1         A#x1      -> A#y2,
A#y2 -> A#returns,  1         -> A#returns,
B#y1 -> A#y1,        A#returns -> B#z1,
B#x1 -> A#x1,        B#x1      -> A#y2,
A#y1 -> A#y2,        2         -> B#z1
```

Ignoring flows from constants, and applying transitivity, the following additional flows are found:

```
B#x1 -> A#returns,  B#x1 -> B#z1,
B#y1 -> B#z1,       A#x1 -> A#returns,
A#y1 -> A#returns,  A#y1 -> B#z1,
A#x1 -> B#z1,       B#y1 -> A#returns
A#y2 -> B#z1,
```

Resulting Formulas

(Same in Infix standard as in Prefix Boyer-Moore):

```
lteq(1,B#x1),          lteq(2,B#y1),
lteq(B#y1,A#y1),       lteq(A#x1,A#y2),
lteq(A#y2,A#returns), lteq(1,A#returns),
lteq(B#y1,A#y1),       lteq(A#returns,B#z1),
lteq(B#x1,A#returns), lteq(B#x1,B#z1),
lteq(B#y1,B#z1),       lteq(A#x1,A#returns),
lteq(A#y1,A#returns), lteq(A#y1,B#z1),
lteq(B#x1,A#x1),       lteq(B#y1,A#returns),
lteq(A#y1,A#y2),       lteq(A#x1,B#z1),
lteq(B#x1,A#y2),       lteq(A#y1,B#z1)
lteq(2,B#z1),
```

SECTION 5

THEOREM PROVER EVALUATIONS

Once either STOS or STOF has produced a formulas file, it must be integrated with a theorem prover to determine whether the generated statements are true or false. One advantage of STOF is that it will be designed to adapt with various theorem provers while STOS is restricted to the theorem provers used by HDM. The STOF verification approach offers users flexibility in choosing a theorem prover.

Three existing theorem provers are good candidates for use with STOF. These are Boyer-Moore, Shostak, and COMPUSEC.

5.1 BOYER-MOORE

The Boyer-Moore theorem prover [3] is currently being used with the standard HDM toolset. Characteristics of this existing theorem prover include:

- Implemented in Interlisp
- Notation for input is lambda calculus in prefix format
- Runs in a fully automatic state (user cannot guide proofs during run-time--does not provide interactive aid to finding proofs)
- Handles induction, propositional logic, user-defined recursive functions
- Report generation is clear and well-organized

5.2 SHOSTAK

The Shostak theorem prover has been developed for use with the Enhanced-HDM toolset. Features of this theorem prover include:

- Implemented in MACLISP
- Notation for input is predicate calculus
- Runs in a fully automatic state, but is user guided (requires an instruction queue)
- Handles propositional logic user-defined recursive functions--will handle induction in the future

5.3 COMPUSEC

The COMPUSEC theorem prover is still under development; however, the majority of the tool has already been implemented and tested. COMPUSEC has conducted an in-house comparison on the operation of the Boyer-Moore theorem prover and the COMPUSEC theorem prover. The COMPUSEC theorem prover was found to require substantially less computational time and resources than the Boyer-Moore theorem prover when proving identical theorems. The following characteristics are either already implemented in the COMPUSEC theorem prover or will be implemented in the near future:

- Implemented in VAX-Pascal
- Notation for input is standard predicate calculus in infix format
- User has option of running theorem prover in an interactive state or a fully automatic state
- Currently handles propositional logic - will handle induction and user-defined recursive functions in near future
- Faster computation time than Boyer-Moore
- Report generation will be clear and well-organized - can be tailored to user's needs

SECTION 6

SOURCE CODE VERIFICATION TOOL SPECIFICATIONS

6.1 STOS CODE VERIFICATION TOOL

6.1.1 SOURCE Subset Handled By STOS

Programming language power and flexibility may conflict with clear representation of information flow. Some language constructs (such as pointers) can cause information flow that cannot be anticipated or derived from static examination of the source. Indeed, most languages include a mechanism for inserting comments into the source code for clarification of such ambiguities. However, such comments are only programmer aids and do not represent valid input for a formal specification. Additionally, program source can be obscure if the source language does not formally specify the actions resulting from a given construct. An example can be found in Ada multi-tasking: multi-tasking is a feature of the language, but a description of how memory allocation is handled during processing of this feature is not explicitly stated. In such cases it is not possible to derive accurate formulas specifying information flow and correctness. Hence, in translation of a source to formulas, the source language must be restricted to an unambiguous subset that can be formally stated.

To determine which constructs of the Ada language will be included in the Ada subset, each construct must be analyzed for its inherent security properties. Constructs that are not well-defined with respect to processing and memory allocation are not suitable for STOS. Such constructs generate information flows that cannot be definitively specified with formulas. If non-deterministic information flows were allowed in the formal verification process, any resulting proofs would also be non-deterministic. Examples of constructs that fall into this category are "exception," "generic," and "task."

- <u>Description</u>

  Certain criteria must be met in order to define an Ada subset that is both functional and verifiable. Any constructs that present problems in formula generation must be either omitted or restricted in the subset. Naturally, restricting the use of a programming language in this way will inhibit some of its usefulness in writing applications. Care must be taken to ensure that the verifiable subset does not significantly limit programming applications.

During lexical analysis all Ada reserved words are recognized; however, only those contained in the subset are processed. Messages will be generated to flag each encountered Ada construct that is not part of the specified subset (see Table 6-1).

- Specification

Following is a specification containing all Ada constructs that are included in the STOS SOURCE subset. All punctuation existing in the Ada language will be included within the Ada subset. This includes binary operators (i.e., +, /, -, and *).

- <blocks> - These constructs act as headers used to group specific portions of the code. They define the processing bodies for the program.

```
begin
body
end
package
use
with
```

- <declarations and types> - These constructs provide a means for defining different entities. The type of an item dictates which operations are permitted on that item.

```
array                       out
at                          procedure
constant                    range
delta                       record
digits                      renames
function                    separate
in                          subtype
is                          type
of
```

- <operators> - These constructs represent specific functions that are to be performed on associated entities.

```
abs                         or
all                         rem
and                         reverse
mod                         xor
not
```

Table 6-1. Ada Reserved Words Handled By STOS

*abort	begin	case	*declare	else
abs	body	constant	*delay	elsif
*accept			delta	end
*access			digits	*entry
all			do	*exception
and				exit
array				
at				
for	*generic	if	*limited	mod
function	*goto	in	loop	
		is		
*new	of	package	*raise	*select
not	or	*pragma	range	separate
null	others	*private	record	subtype
	out	procedures	rem	
			renames	
			return	
			reverse	
*task	use	when	xor	
*terminate		while		
then		with		
type				

* Not member of verifiable Ada subset

- <statements> - These constructs combine to form a list of all the possible Ada statements that can be analyzed using the STOS approach.

case	loop
do	null
else	others
elsif	return
exit	then
for	when
if	while

6.1.2 SPECIAL Subset Used By STOS

HDM is an aid to design, implementation, and verification of software systems. It includes the language SPECIAL, the theorem prover, and the MLS formula generator used for multilevel security. HDM has been used widely in the verification of software currently in use by the Department of Defense.

Specifications of a multi-level secure (MLS) system can be written in SPECIAL [21]. A SPECIAL specification provides a description of the external visible behavior of a system (i.e., a description of how the system responds to each possible external stimulus). Possible external stimuli are defined as the invocation of the visible operation references (a visible operation together with a particular set of values for its arguments). The specification describes how the internal state of the system changes when a particular visible operation reference is invoked and identifies the value returned by the invocation of the operation reference.

MLS models require that there be a set of values, L, which act as security levels and that these values be partially ordered under some binary relation, named here as "lteq". This information must be provided before a proof can be attempted.

Applying the MLS proof tools yields a listing of the attempted proofs of a set of formulas. If the attempted proofs of all the formulas are successful, this implies that the specification is MLS with respect to the given security levels. Note that the proof does not determine how the security levels are interpreted or how access to them is controlled in a given implementation.

SPECIAL was originally designed as a vehicle for specifying system development using a top-down approach. It is a broad language consisting of many constructs that are intended to represent functionality and/or information flow. When using SPECIAL as a vehicle for formal verification, information flow is the primary concern. Constructs dealing solely with functionality are unnecessary. In fact, only a subset of the SPECIAL language is supported by the HDM MLS tool. These factors render the use of an unabridged SPECIAL both inefficient and unnecessary in the STOS approach. Therefore, a subset of the language must be specified for use with STOS.

- Description

  A module specification in SPECIAL consists of six paragraphs, each of which is optional in a given specification. In its most general form, the top-level structure looks like:

```
MODULE <symbol>

   TYPES
      <types body>

   PARAMETERS
      <parameters body>

   DEFINITIONS
      <definitions body>

   EXTERNALREFS
      <externalrefs body>

   ASSERTIONS
      <assertions body>

   FUNCTIONS
      <functions body>

END_MODULE
```

  - The TYPES paragraph contains the declarations for all internal named types (including designator types) used in the module specification. The only two types which are necessary for tracing information flow follow.

    a. "LABEL" represents the security label of subjects and objects within the specification. LABEL can be represented as either a designator or as an aggregate of designators (one designator for each component of the security label).

    b. "DATA" is used to represent a single object. Structured objects are split into their component parts when represented in FTLS, with each component being represented by a distinct object.

- The PARAMETERS paragraph contains the declarations for symbolic constants called module parameters. Module parameters are similar to V-functions (defined in the FUNCTIONS paragraph) except that their values cannot be changed. These symbols are used to represent either information flow out of one or more objects or constants whose properties are defined by assertions rather than function definitions. [12]

  A module parameter that must appear in all FTLS used for security analysis is "lteq" (less than or equal to). The format used is lteq(DATA L1,L2). The format of "lteq" is defined in the PARAMETERS paragraph, and its actual effects are defined in the ASSERTIONS paragraph. "lteq" determines, by comparing security labels of objects L1 and L2, if information flow is allowed from L1 to L2.

  The following module parameters were developed by COMPUSEC specifically for use in security analysis. The first two are symbolic constants, the rest are parameters used for specifying information flow.

  a. K! - represents any unlabeled constant. In almost all cases constants are unimportant to the security aspects of information flow, and hence are represented by a value which will be ignored by the MLS tool.

  b. LLLLL - represents any unknown security label. No assertions are made in the FTLS about the state of LLLLL and hence if an MLS analysis is made on FTLS containing LLLLL, there will be a security flaw shown at every appearance of an LLLLL.

  c. Fn (i.e. F1, F2...Fn) - is a place holder for n objects. For example, an entry in the PARAMETERS paragraph would have the form:

     ```
     F3(DATA V1,V2,V3),
     ```

     where "F3" shows information flow from three objects. When processing the statement

     ```
     Y := Z - X + 42;
     ```

     The information flow within this statement is in no way dependent on the arithmetic operations, and hence the FTLS would contain the statement

     ```
     'Y = F3(Z,X,K!);
     ```

d. CONDn - behaves exactly like Fn. "COND" is used when generating FTLS for conditional statements, and is used instead of 'F' for readability purposes. An example of its use is the translation of the statement

```
if X < Y then
   Z := X;
```

This will be represented in the FTLS by

```
COND2(X,Y) =>
    ('Z = F1(X));
```

- The DEFINITIONS paragraph contains the definitions for macro-like auxiliary function definitions. A common use of the DEFINITIONS paragraph is to represent a complex structured entity using a single symbol. For example, a three compartment label can be represented by a single string if the following definition is made:

  ```
  LEVEL C_0_0 IS LABEL(CC,00,CC);
  ```

  (LABEL would have to have been defined in the PARAMETERS paragraph as type LEVEL, and LEVEL defined as STRUCT_OF(DATA C1,C2,C3) where C1,C2, and C3 are the names of the three compartments.)

- The EXTERNALREFS paragraph contains the declarations for other modules that are externally referenced in the specification. These objects include designator and scalar types; V-, O-, and OV-functions (defined in FUNCTIONS paragraph); and parameters.

- The ASSERTIONS paragraph contains assertions that are constraints on the module's parameters, and invariants of the module that need to be proved from its specification. The ASSERTIONS paragraph is used in security FTLS to define the properties of "lteq." These assertions determine which information flows will be allowed during MLS analysis. The truth value of lteq(X,Y) is asserted to be either true or false for pairs of security labels. In the case of compartmented labels, lteq is defined as the conjunction of truth values of relations between corresponding components of the two labels.

The following assertions could be made for uncompartmented labels:

```
lteq(unclassified,unclassified),
{ read as unclassified may flow into unclassified }
lteq(unclassified,confidential),
{ read as unclassified may flow into confidential }
lteq(confidential,secret),
{ read as confidential may flow into secret }
.
.

~lteq(secret,unclassified)
{ read as secret may not flow into unclassified }
.
.
```

If labels have two compartments representing classification and integrity, then two new functions will be needed for comparison of the components:

```
slteq(X,Y)        - used for classification checks
ilteq(X,Y)        - used for integrity check
```

The relationship between lteq and the two functions slteq and ilteq is then asserted in the FTLS by

```
FORALL DATA X1; DATA Y1;
       DATA X2; DATA Y2;
       lteq(LABEL(X1,Y1),LABEL(X2,Y2)) =
                    (slteq(X1,X2) AND
                     ilteq(Y1,Y2));
```

This states that lteq is true if and only if slteq and ilteq are simultaneously true.

- The FUNCTIONS paragraph contains the definitions for all V-, O-, and OV-functions of the module. A VFUN returns a value. An OFUN changes system state. An OVFUN changes system state and returns a value. The actual information flow within the system is represented in this paragraph. The structure of expressions appearing in the FUNCTIONS paragraph was described in the discussions of translation methodology of STOS.

In defining a SPECIAL subset, certain criteria must be met. All constructs in the defined subset must be supported by the HDM MLS tool. Also, the SPECIAL subset must be able to adequately specify, with respect to information flow, the chosen source subset.

The SPECIAL language consists of the keywords found in Table 6-2. The following punctuation is specific to the SPECIAL language:

'	-	new value
=>	-	implies
?	-	undefined
->	-	function return value
$( )	-	comment

Punctuation generic to all languages (i.e., +, -) is included in the SPECIAL language. It performs the same basic functions as in other programming languages. However, this punctuation is not specified as part of our subset and therefore will not be detailed in this report.

o <u>Specification</u>

SRI International, the developers of the HDM toolset, placed certain restrictions on SPECIAL for use with HDM. These restrictions, as stated below, form a baseline for our subset of the language. [22],[23]

- No recursive or mutually recursive definitions are permitted.

- The key words NEW, TYPECASE, and RESOURCE_ERROR may not be used.

- An expression may contain no more than one reference to a new value. A new value is either a quoted V-function reference (i.e. 'identifier), an EFFECTS_OF expression, or, in an OV-function, a return value reference.

- In the effects of an OV-function, the return value reference may occur only once.

- A new value reference may not occur in:
  -- The qualification part of a LET, FORALL, EXISTS, SOME, or set expression.
  -- The antecedent of an implication.
  -- The boolean expression in an "if" expression.
  -- The range of a vector constructor.

- If the specification consists of more than one module, the directed graph of external references between the module must have no loops.

COMPUSEC has extensive practical experience using SPECIAL. From this knowledge base, COMPUSEC has limited the list of useable constructs even further to produce a more concise and effective security analysis language. Also, COMPUSEC has designed some additional constructs in SPECIAL, which aid in the security analysis of the system. These constructs (COND, F, K!, DATA, LEVEL, LLLLL) function as parameters in the language. A description of all SPECIAL constructs [24] and COMPUSEC-developed constructs that combine to form the SPECIAL subset for STOS are found in Table 6-2 and in the following descriptions:

- **<blocks>** - These constructs act as headers used to identify specific parts of the SPECIAL program. The PARAMETERS section is where the COMPUSEC-developed constructs (DATA, LEVEL, COND, F, LLLLL, and K!) are defined. DATA and LEVEL are used to define identifiers which represent security label information. The ASSERTIONS section specifies the security rules by which information flow is analyzed.

  ASSERTIONS
  DATA
  DECLARATIONS
  DEFINITIONS
  EFFECTS
  EXTERNALREFS
  FUNCTIONS
  LEVEL
  MODULE
  PARAMETERS
  TYPES

- **<declarations and types>** - These constructs are used as descriptors of identifiers. They define how a variable will be represented and manipulated within the program.

  BOOLEAN
  DESIGNATOR
  FORALL
  OFUN
  ONE_OF
  OVFUN
  STRUCT
  STRUCT_OF
  VFUN

- **<operators>** - These constructs represent specific functions that are to be performed on associated entities. Of particular interest is the operator F which stands for "function of." This is one of the COMPUSEC-developed SPECIAL constructs. It is a generic symbol to represent the dependency of a "new value" on the identifiers involved in its creation. It addresses information flow for any and all possible operations without concern for algorithmic functionality.

  AND
  COND
  F
  FROM
  IN
  IS
  NOT
  OF
  OR
  WITH
  =>
  ?
  ->

Table 6-2. SPECIAL Constructs Handled By STOS

and	boolean	*cardinality	declarations	effects_of
assertions		*char	definitions	effects
*assert			*delay	else
			derivation	end
			designator	end_if
			*diff	*end_map
				end_module
				*exceptions
				*exceptions_of
				exists
				externalrefs
false	hidden	if	*let	*map
*for		in	*length	*mappings
forall		initially		*max
from		*inset		*min
functions		*inter		*mod
		*integer		module
		*invariants		
		is		
*new	of	parameters	*real	*set_of
not	ofun		*resource_error	*some
	*on			struct
	one_of			struct_of
	or			*subset
	ovfun			
then	*undefined	*vector	*with	
*to	*union	*vector_of		
true	*until	vfun		
types				
*typecase				

* Not member of verifiable Ada subset

- <identifiers and expressions> - These constructs are a means of specifying different identifiers and expressions in the language. For example, all constants are represented as K!. LLLLL is a place holder for an unknown label.

FALSE	LLLLL
K!	TRUE

- <statements> - These constructs identify specific types of statements in SPECIAL. Each time the value of a variable is modified, it is represented by a "new value" (') statement.

DERIVATION	EXISTS
EFFECTS_OF	IF
ELSE	INITIALLY
END	THEN
END_IF	'
END_MODULE	

- <visibility rules> - The HIDDEN construct restricts referencing of VFUNs. When used, only the module in which the VFUN exists may reference the VFUN.

  HIDDEN

6.1.3 STOS Cross-Compiler Specification

The subset of the Ada language described in the previous section is input to STOS for translation. STOS will output a FTLS that captures all information flow represented in the Ada source code. STOS will also label the FTLS so that it can be submitted to HDM's MLS tool for generating verification conditions. Output from the MLS tool can then be submitted to the theorem prover.

The components of STOS are:

- Ada Security Analyzer
  - Lexical Analyzer
  - Parser
- Labeler
- Propagator
- SPECIAL generator
- Automated Tracer

These components combine to form the cross compiler. Each phase will be specified in subsequent sections.

Figure 6-1 depicts a high level description of the component steps in the STOS cross-compiler. Subsequent sections describe each component in further detail. Required functionality is specified for each component. Details of implementation (i.e. choice of program source language, type of parser) are left to the implementor.

6.1.4 STOS Ada Security Analyzer

- <u>Lexical Analyzer</u>

  The Lexical Analyzer (see Figure 6-2) acts as the interface between the source program and the parser. It reorganizes Ada source code into a format that is readable by the parser. During this phase, the Ada source is modified by introducing tokens to the code.

  - <u>Description</u>

    The Lexical Analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers, keywords, constants, operators, and punctuation symbols (i.e., commas and parentheses) are typical tokens. For example, the following Ada statement contains seven tokens:

Ada	Tokens
if (I = MAX) then	if
	(
	I
	=
	MAX
	)
	then

    In general, each token is a substring of the source program and is to be treated as a single unit (it is not reasonable to treat M or MA of the identifier MAX in the example above as an independent entity). There are two kinds of tokens: specific strings such as "if" or a semicolon, and classes of strings such as identifiers, constants, or labels.

    The Lexical Analyzer can also be used to customize compilation in order to reflect application or security requirements. It can retrieve and insert code referenced by

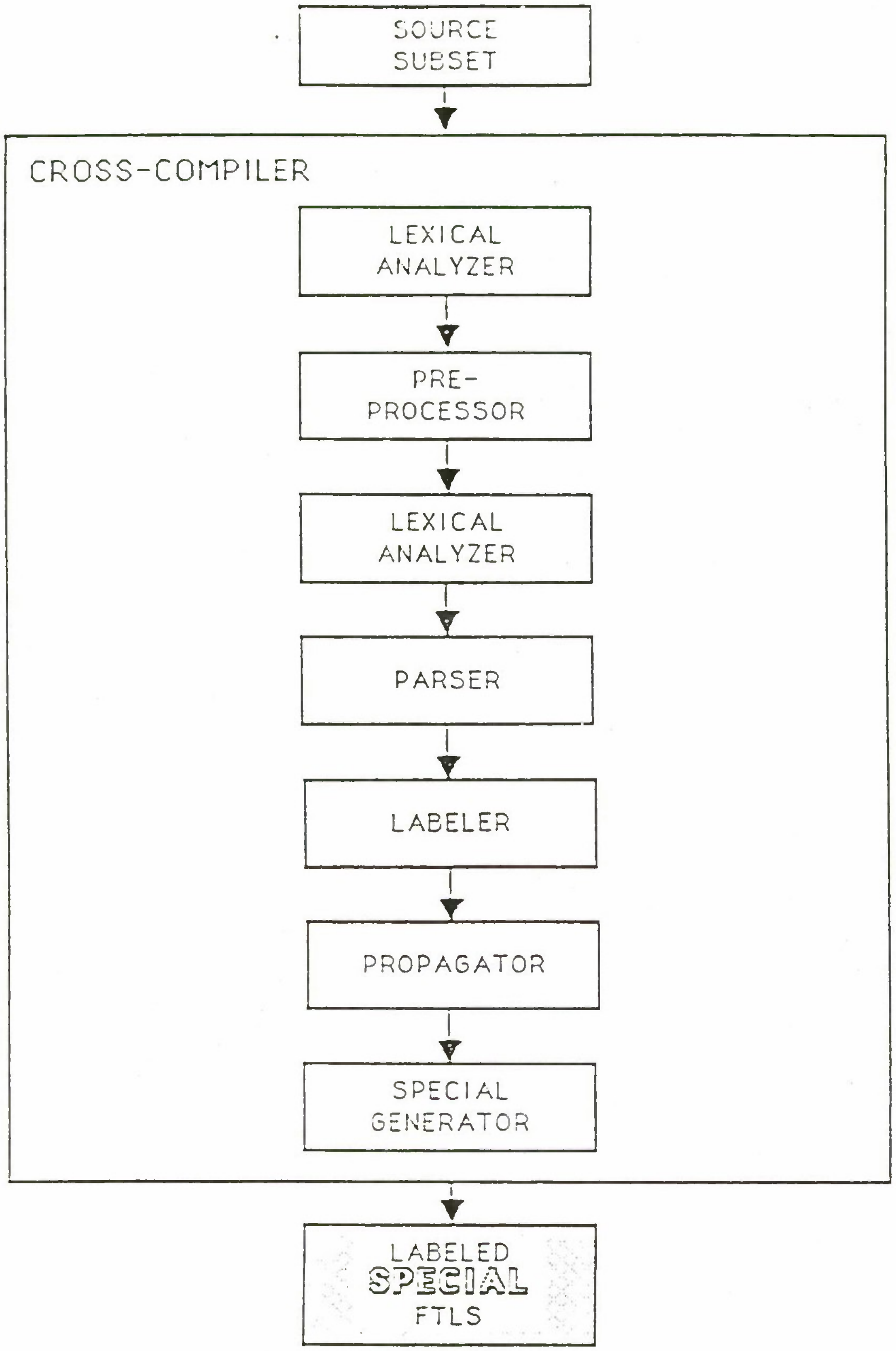


Figure 6-1. STOS Cross-Compiler

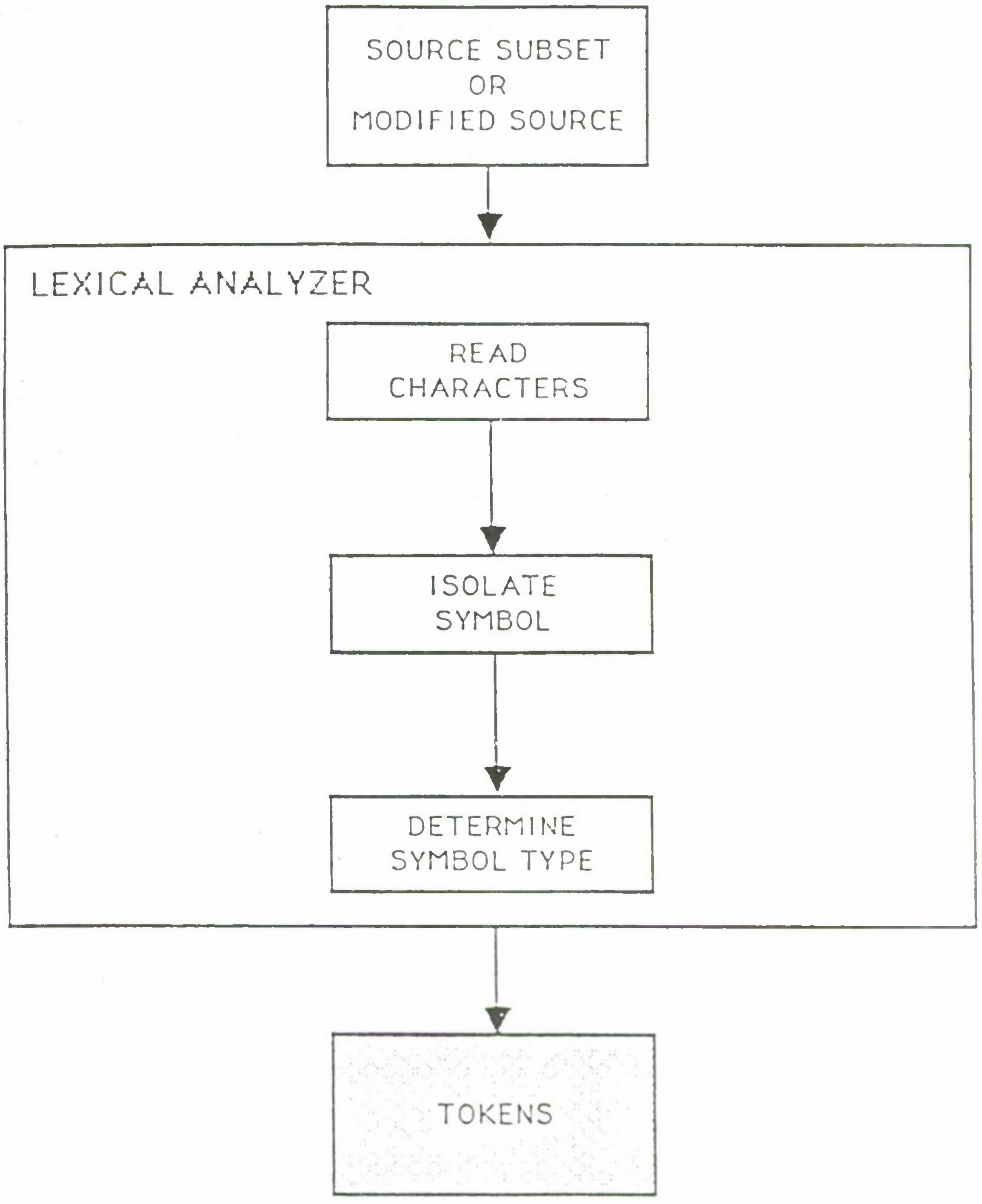


Figure 6-2. STOS Lexical Analyzer

Ada "include" statements. It can also capture information flow contained in externally referenced files such as library routines, and can comment out certain routines determined to have no security relevance. It can resolve user definitions. Finally, the Lexical Analyzer can format system requirement comments in order to facilitate automated requirements traceability (see Section 6.1.8).

- Specification

The Lexical Analyzer will take Ada source code as input and return tokens as specified by the syntax given in ANSI-MIL-STD-1815A [9]. The Lexical Analyzer shall read a single character at a time from the input Ada source code. The Lexical Analyzer will then identify and isolate symbols from this character stream by following lexical rules for detecting symbol boundaries. As a symbol is found it is looked up in the stored symbol table. When a symbol match is found in this table, the value associated with the symbol shall be used to determine symbol type.

The token shall be returned in two parts: The first part represents the token value, the second represents the token type. Token values can be ASCII values for strings, numbers, symbols, or null. Token types can be constant, identifier or the valid Ada delimiters, keywords, and operators.

Token values shall be: ASCII or numeric value for constants, ASCII string value for identifiers, ASCII value for delimiters and operators, Null for reserved words.

- Parser

The parser serves two primary functions. First, it checks that the tokens appearing in its input occur in patterns that are permitted by the specification for the source language. Second, it creates a tree-like representation of the input using these tokens (see Figure 6-3). The generated parse tree is used in subsequent phases of the STOF cross-compiler.

  - Description

  Input tokens are checked for syntactic and semantic correctness before the parse tree is created. Syntax and semantic errors can therefore be flagged. The Ada expression in the following example will generate an error when it is evaluated by the parser:

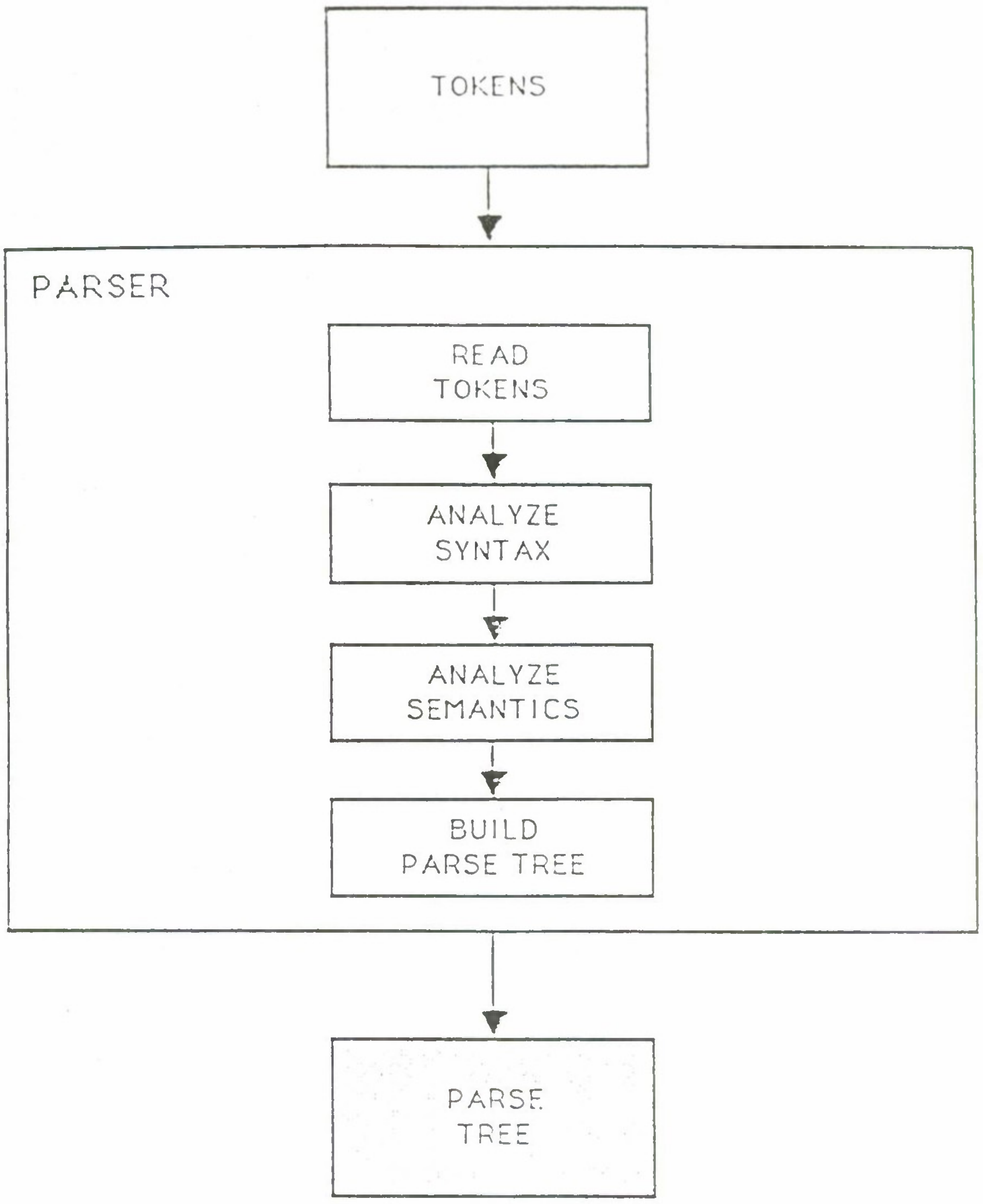


Figure 6-3. STOS Parser

Ada	Parser Input
A + / B	A-identifier
	+-operator
	/-operator
	B-identifier

The parser's syntax analyzer will detect an error when it receives the /-operator token, because the presence of two adjacent binary operators violates Ada expression formation rules.

The parser will also analyze Ada type declarations and enter type information into the symbol table. Type information is necessary to determine a statement's semantic consistency. In Ada, a single data name may be used to reference several distinct data items declared in different scopes. The symbol table is used to distinguish different contexts where the same data item names are used.

Although the parser does not need to allocate space for different types, it must match types and maintain unique names. Context-dependent Ada data item names must be changed to unique names that can be used throughout the entire translation. One way to accomplish this is to append a number to each declared instance of a name.

The generated parse tree must show the hierarchical structure of the incoming token stream. This is accomplished by grouping together tokens from the token stream. Token groupings will then reflect processing sequence according to the semantics of the Ada language specification.

- <u>Specification</u>

The parser shall accept tokens generated by the Lexical Analyzer. Type information shall be extracted from tokens and entered into the symbol table. Unique names shall be established and maintained for every data item even when the same name is used in different contexts. Tokens and unique names shall be grouped to reflect the hierarchical structure of the input source code. A parse tree is constructed from this source. Initial and final conditions for each statement are stored at the same point in the tree as the statement. The output parse tree shall be suitable for use in subsequent phases of the cross-compiler.

6.1.5 STOS Labeler

Labeling is a necessary phase in the verification of an MLS system. It serves to identify objects and subjects that require special handling for security by associating a label classification. It also allows an information flow analysis tool to identify flows that can potentially compromise classified information (see Figure 6-4).

- <u>Description</u>

  The Labeler reads the static label data base from disk and stores it in a format suitable for high speed label retrieval. The Ada parse tree is input to the labeler with data items in the tree having an empty slot for a label. For each data item in the parse tree, the labeler will search the static label data base for the corresponding label. If the label is found, the labeler will insert the label from the static data base into the empty slot. If the label is not found, the slot will be filled with an unspecified floating label.

  In this manner, the labeler will fill the parse tree with static and unspecified floating labels.

- <u>Specification</u>

  The Labeler will take as input a data base of static label information and the parse tree as generated by the STOS parser. It shall label data items in the parse tree with labels from the data base of static labels.

  Data items will be tuples composed of a data name and label. The parse tree from the Ada STOS parser will be a structure delineating hierarchically the operations and data items of the source with floating unspecified labels.

  The static label data base will be composed of data items with static labels; that is, data items with known unchanging labels.

  The labeler's primary action will be to associate entries in this data base with entries found in the parse tree. It shall then label every data name in the parse tree with the corresponding data base entry label.

  This action shall serve to label all statically labeled data items in the parse tree. All unlabeled data items will default to floating labels to be set later by the propagator.

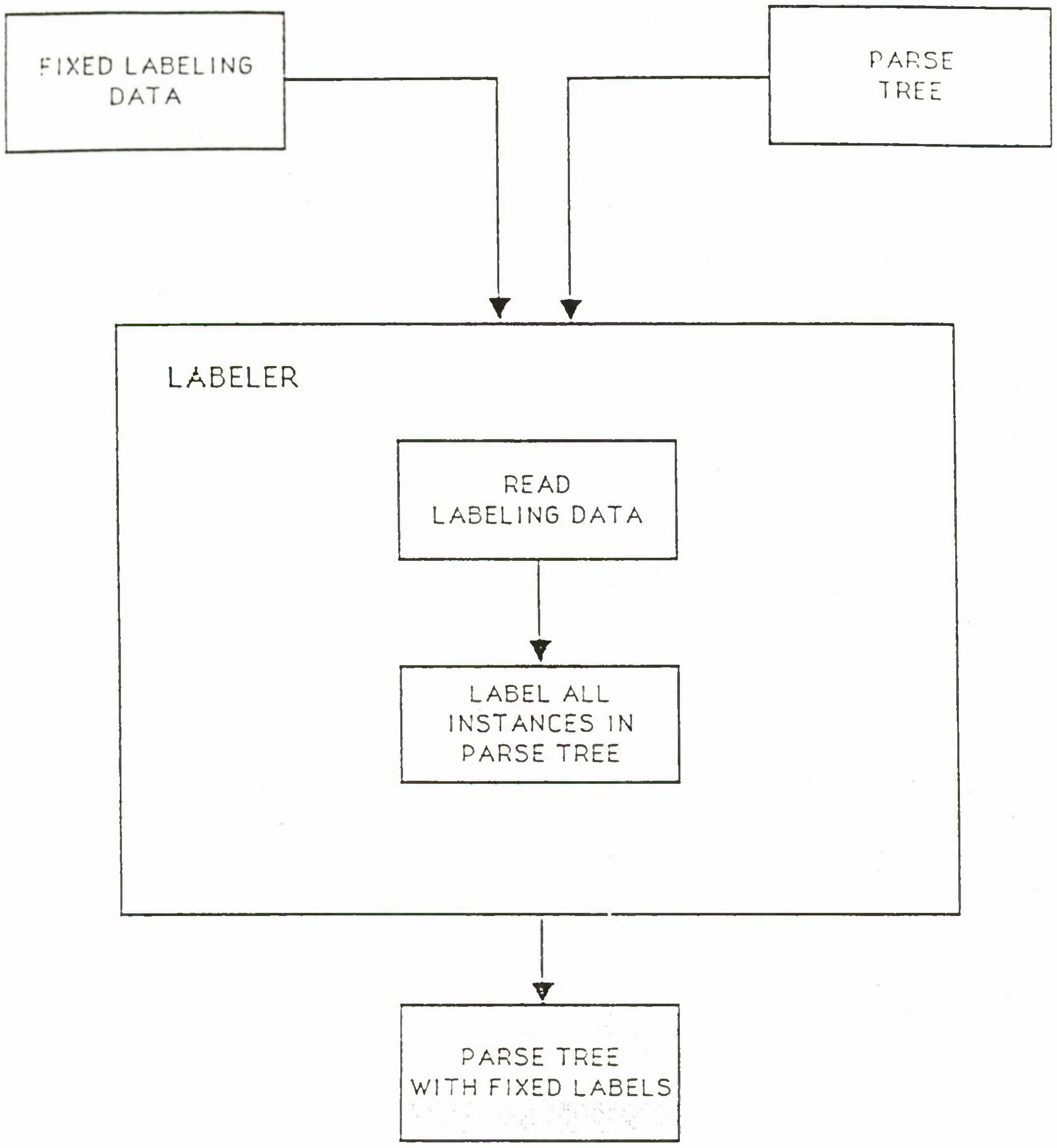

FIXED LABELING DATA
PARSE TREE
LABELER
READ LABELING DATA
LABEL ALL INSTANCES IN PARSE TREE
PARSE TREE WITH FIXED LABELS

Figure 6-4. STOS Labeler

6.1.6 STOS Propagator

Label propagation is the process by which labeling information is transferred from data items with static labels to data items with floating labels. This happens when information from the static item is passed to the floating item. If the static label is higher than the floating label, the floating data item will contain information from the static item. Therefore, it should also have this dominant label. Thus, in this phase, unlabeled and floating data items are labeled with the highest label of the information they will contain (see Figure 6-5).

- <u>Description</u>

  On input of the Ada parse tree, the propagator's task is to determine the dependencies of unspecified floating labels on statically labeled data items. The propagator will then change these unspecified labels to the most dominant label of the data items on which they depend. To do this, the propagator uses knowledge about labels and their relationships. It must also have knowledge about dependencies shown in the Ada parse tree, and in the static labels set by the labeler.

  Labeling conflicts may occur during label propagation. The propagator must have facilities for reporting errors resulting from conflicts between static and floating labels. This output should be able to reference data items as shown in the Ada source by line numbers showing the correct data name.

- <u>Specification</u>

  The propagator will take as input an Ada parse tree generated by the STOS Ada parser. This parse tree will indicate information flow and dependencies through a hierarchical structure.

  Data items given floating unspecified labels in the parse tree will be collected. For each of these data items, a dependency list will be established. This list will show dependencies between data items with floating labels and all other data items that pass information to them.

  For every data item with a floating label, the propagator will determine the most dominant label in its dependency list and will set the floating label to that label. It will repeat this process until all floating labels are given the most dominant label in their dependency lists. Any resulting conflicts will be reported.

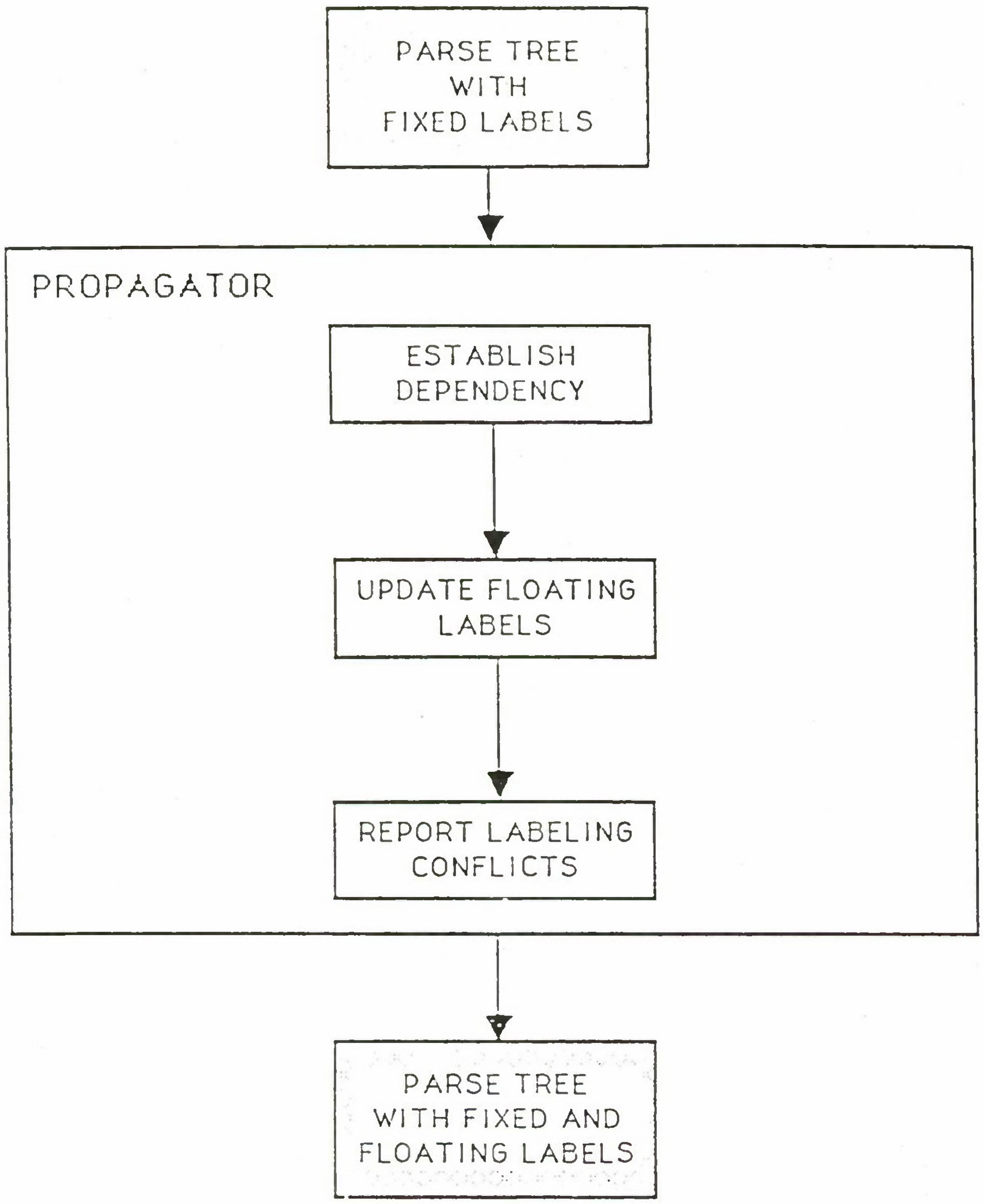

PARSE TREE WITH FIXED LABELS
PROPAGATOR
ESTABLISH DEPENDENCY
UPDATE FLOATING LABELS
REPORT LABELING CONFLICTS
PARSE TREE WITH FIXED AND FLOATING LABELS

Figure 6-5. Propagator

6.1.7 STOS SPECIAL Generator

The SPECIAL generation phase converts a labeled Ada parse tree into a sequence of SPECIAL instructions representing information flow. Ada Assignment statements, conditionals, loops, procedures, and functions represented in the parse tree will be translated into valid SPECIAL statements (see Figure 6-6.)

- Description

  The SPECIAL generator will essentially reduce source code to a high level formal description of source code information flow. Translation rules may need to be tailored for applications where conditions exist that affect theorem proving. For example, applications that take advantage of known hardware security features may not reflect this in their source code. In any case, certain general translation rules can be specified and followed. In addition, some optimization could be performed during SPECIAL generation in order to remove any redundant information flows.

- Specification

  Input to the Special generator is the parse tree generated by the STOS parser. Output from the special generator is the special translation of the Ada source. As a minimum, the following set of translation rules will be applied to the input parse tree:

  - Assignment Statements

    a. Every term on the lhs of an assignment statement will cause a NEWVALUE statement to be generated that is a function of every term on the rhs of the assignment.

    b. The assignment of a variable of structured type to another variable of the same structured type will produce as many NEWVALUE statements as there are scalar types making up the structured type.

    c. If a variable of structured type has array type components, then any identifiers that are used to index the array are extracted. Information flow is shown from the array indices on either side of the assignment statement to the variable on the lhs of the assignment.

  - Conditionals

    a. Information flows will be shown from all identifiers appearing in the condition part of the "if" or "else if" statement to all "outputs" of any statements appearing in the "then", or "else" part of the "if" statement. All

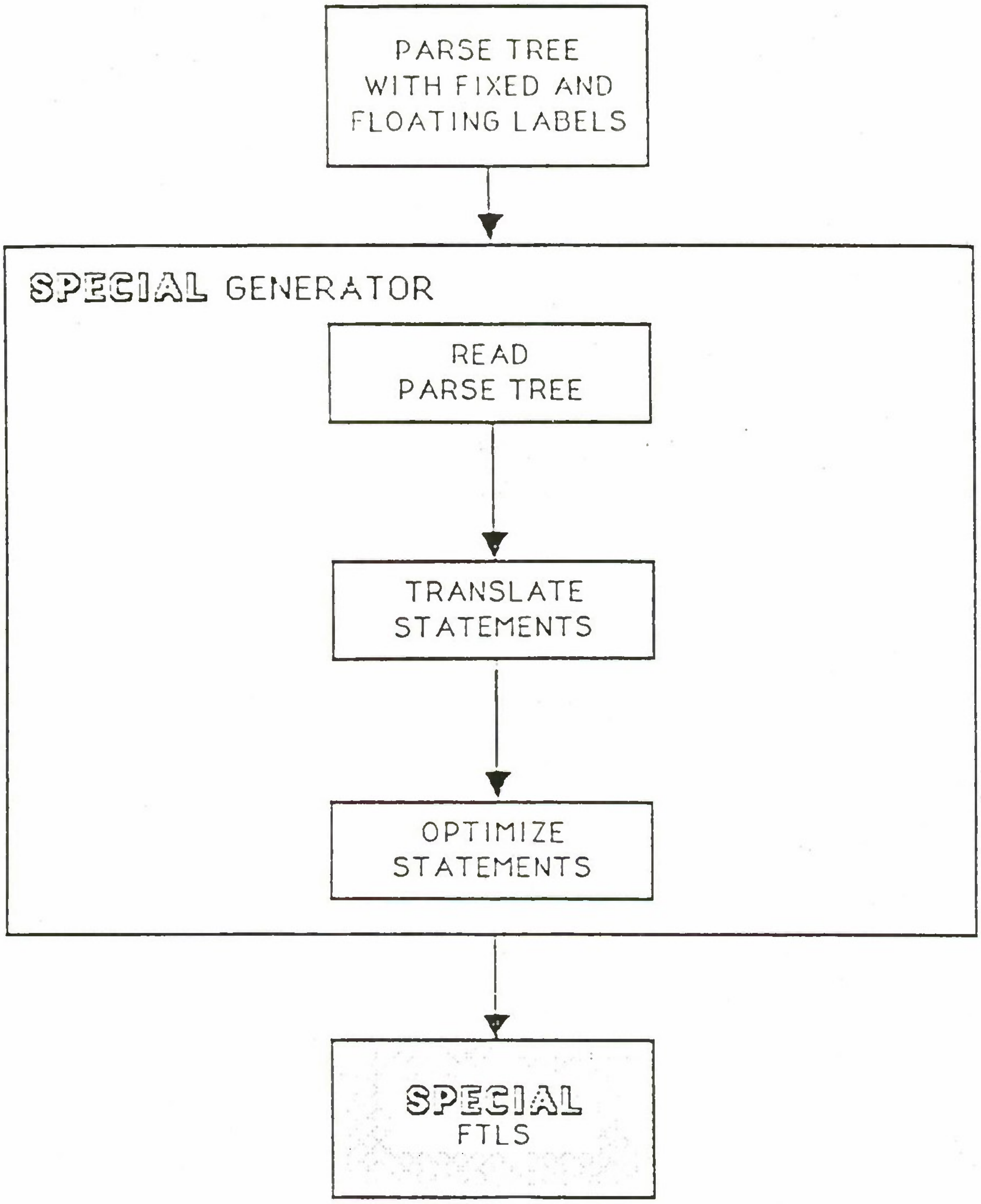


Figure 6-6. SPECIAL Generator

identifiers appearing in the condition are collected, and an implication statement is generated with these identifiers placed on the lhs, "while" all statements within the scope of the "if" will be placed on the rhs.

b. The "case" construct will be represented as a series of "if" statements having the same conditional expression.

- Loops

  - If the loop is named, it will be extracted and stored. If there are no conditions on the loop then statements within the loop will be processed as if the loop did not exist. "If" statements leading to an unconditional exit will be flagged.

  - Any identifiers appearing in the iteration scheme of a "for" statement, existing in the conditional part of a "while" statement, or representing conditions for loop exit will be collected and put on the lhs of an implication, with the rhs representing the translation of any statements within the loop.

- Parameters and Function "RETURN" Statements

a. "In" parameters will cause a NEWVALUE statement to be generated prior to the EFFECTS_OF statement that shows flow from the actual parameter to the new name generated for the procedure's formal parameter list.

b. "Out" parameters will cause a NEWVALUE statement to be generated following the EFFECTS_OF statement that shows flow from the formal parameter to the actual parameter.

c. "In out" parameters will produce both of the above results.

d. If a subprogram returns a value, a new name for the returned value will be made by concatenating the subprogram name with the phrase "returns". This new name will be used in any place that the function value is used. An EFFECTS_OF statement will be placed preceding the use of the new name

6.1.8 STOS Automated Tracer

It is desirable to trace system requirements through progressive software design stages and into the implemented code. Establishment of correspondence between requirements and code simplifies the problem of demonstrating that a system is ready to be deployed. Automating this traceability could decrease the cost of certifying a software system. [24],[17]

- Description

  The STOS cross-compiler could be used to support automated system requirement traceability. The pre-processor could format comments that contained requirements identifiers so that the comment also contained the name of the task or procedure in which it occurred. This modified source would be suitable for use as input to the tracer.

- Specification

  Tracer inputs shall be a suitably commented Ada source code module and a numerically organized system requirements list. Ada source code input shall include comment statements for tasks or procedures that contain a requirement identifier referencing the relevant system requirement.

  For each numbered requirement in the requirements list, the tracer shall search the Ada source code comments for a matching requirement identifier. If a match is found, the tracer shall identify the task or procedure containing the comment along with the source code line number. If no match is found, then the tracer will output the requirement number followed by an error message.

6.2 STOF CODE VERIFICATION TOOL

A subset of the Ada language can be input to STOF for translation (see Section 6.3.1). STOF will output a file of formulas that captures all information flow (as specified in Sections 4.2 through 4.2.3). In addition to generalizing formulas about information flow, STOF can generate a formula file representing proof of correctness properties found in the Ada source code (see Section 6.3.2). Either one or both formulas files can be input to a theorem prover.

Proof of correctness for programs is discovered by using predefined axioms to repeatedly rewrite formal logical statements about the program until the statements can be proven true. Failing to prove a statement true does not always mean that the program is incorrect--it may be the result of a lack of information in the logical statements about the program, or insufficient power in the theorem prover.

The major difference in the approaches to MLS analysis and proof of correctness is the kind of information that is primarily used. In the case of MLS analysis, information flow is the predominant consideration. In proof of correctness, control flow and state information are the most important.

The components STOF are:

- Ada Security Analyzer
  - Lexical Analyzer
  - Parser
- Security Graph Generator
- Verification Condition Generator (VCG)
  - MLS Flow Analyzer
  - Proof of Correctness Analyzer
  - Formatter
- Environment Support Tool

Figure 6-7 shows the STOF cross-compiler. Individual components will be described in the following sections, and required functionality will be specified for each component. Details of implementation (i.e., choice of program source language, type of parser) are left to the implementor.

6.2.1 SOURCE Subset Handled By STOF

Programming language power and flexibility may conflict with clear representation of information flow. Some language constructs, such as pointers, can cause information flow that cannot be anticipated or derived from static examination of the source. Indeed, most languages include a mechanism for inserting comments into the source code for clarification of such ambiguities. However, such comments are only programmer aids and do not represent valid input for a formal specification. Additionally, program source can be obscure if the source language does not formally specify the actions resulting from a given construct. An example can be found in Ada multi-tasking: multi-tasking is a feature of the language, but how memory allocation is handled during processing of this feature is not explicitly stated. In such cases it is not possible to derive accurate formulas specifying information flow and correctness. Hence, in translation of a source to formulas, the source language must be restricted to an unambiguous subset that can be formally stated.

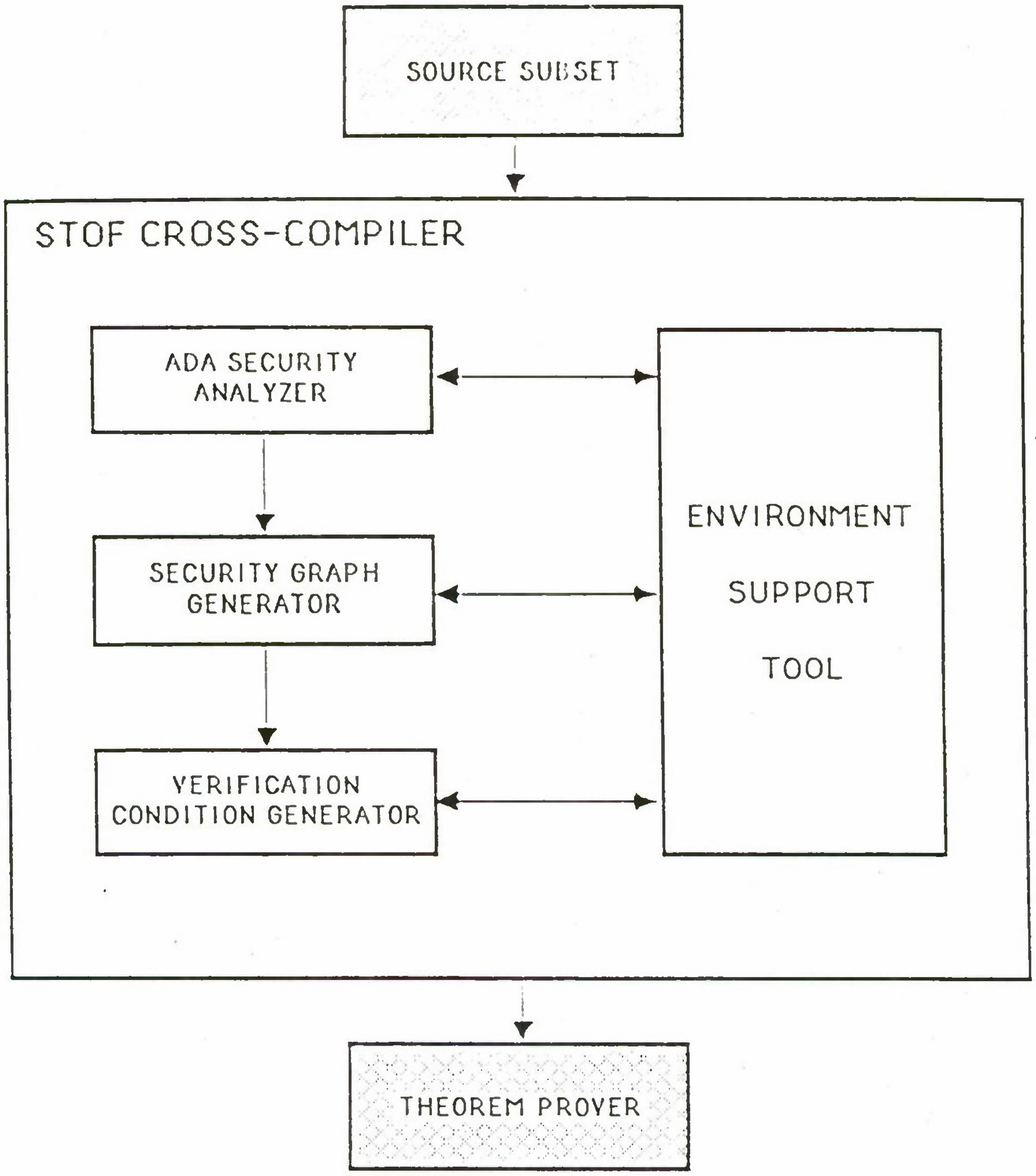


Figure 6-7. STOS Cross-Compiler

To determine which constructs of the Ada language will be included in the Ada subset, each construct must be analyzed for its inherent security properties. Constructs that are not well-defined with respect to processing and memory allocation are not suitable for STOF. Such constructs generate information flows that cannot be definitively specified with formulas. If non-deterministic information flows were allowed in the formal verification process, any resulting proofs would also be non-deterministic. Examples of constructs that fall into this category are "generic" and "task."

- Description

  Certain criteria must be met in order to define an Ada subset that is both functional and verifiable. Any constructs that present problems in formula generation must be either omitted or restricted in the subset. Naturally, restricting the use of a programming language in this way will inhibit some of its usefulness in writing applications. Care must be taken to ensure that the verifiable subset does not significantly limit programming applications.

  During lexical analysis all Ada reserved words are recognized; however, only those contained in the subset are processed. Messages will be generated to flag each encountered Ada construct that is not part of the specified subset (see Table 6-3).

- Specification

  Following is a specification containing all Ada constructs that are included in the subset. All punctuation existing in the Ada language will be included within the Ada subset. This includes binary operators (i.e., +, /, -, and *).

  - **<blocks>** - These constructs act as headers used in order to group specific portions of the code. They define the processing bodies for the program.

begin	package
body	use
declare	with
end	

Table 6-3. Ada Reserved Words Handled By STOF

*abort	begin	case	declare	else
abs	body	constant	*delay	elsif
*accept			delta	end
*access			digits	*entry
all			do	exception
and				exit
array				
at				
for	*generic	if	*limited	mod
function	goto	in	loop	
		is		
*new	of	package	raise	*select
not	or	*pragma	range	separate
null	others	*private	record	subtype
	out	procedures	rem	
			renames	
			return	
			reverse	
*task	use	when	xor	
*terminate		while		
then		with		
type				

* Not member of verifiable Ada subset

- **<declarations and types>** - These constructs provide a means for defining different entities. The type of an item dictates which operations are permitted on that item.

array	of
at	out
constant	procedure
declare	range
delta	record
digits	renames
exception	separate
function	subtype
in	type
is	

- **<operators>** - These constructs represent specific functions that are to be performed on associated entities.

abs	or
all	rem
and	reverse
mod	xor
not	

- **<statements>** - These constructs combine to form a list of all the possible Ada statements that can be analyzed using STOF.

case	loop
do	null
else	others
elsif	raise
exit	return
for	then
goto	when
if	while

6.2.2 Proof Of Correctness Formulas

With STOF, the Ada subset must be translated into formulas that can be analyzed for both MLS and proof of correctness properties. In the specification that follows, the formulas are written in infix format for ease of understanding and explanation. However, if desired, the formulas could be represented in other formats as well.

- Description

The following specification describes the statements in the subset of Ada appropriate for use as part of an STOF system. A proof rule (as used by STOF) and a graphical representation of each statement exist. The proof rule in each case is represented either as a single expression or as an implication.

Rules that are represented by a single expression are processed directly by the formula generator. Rules that are represented as an implication have both an lhs component and an rhs component. The lhs of the implication is the rule that would be processed by the formula generator, and the rhs of the implication is the way the statement would appear in the source.

The following notations are used in the proof rules [25]:

- P - a logical statement
- {P} - a logical statement that is a condition that must be satisfied. Alternately, a pre-condition.
- {Q} - a logical statement that is a condition that must be satisfied. Alternately, a post-condition..
- **P(N)** - the effects of a call to logical statement P with formal parameter N
- **P(X|Y)** - in the logical statement P, systematically replace all occurrences of X by Y
- **P(N)(N|A)** - the effects of a call to P with the formal parameter N replaced by the actual parameter A
- **r(F)** - a return statement of a function with the name F
- S - results of processing the program statement(s) represented by S
- H - attribute and axiom information derived from declarations
- <<L>> - a label named L
- => - implies
- <= - less than or equal to

- <u>Specification</u>

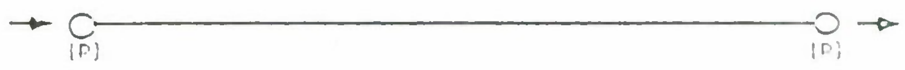
  - **Null Statements**

    The null statement in Ada has no effect, and can be represented by

{P} null {P}

If pre-condition P is true then the post-condition P is true (unchanged).

Graphic Representation:



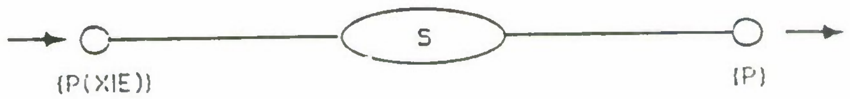
null

- Assignment Statements

    {P(X|E)} X := E {P}

If all occurrences of X are replaced by E in the logical statement P, then the statement, X:= E, is true.

Graphic Representation:



X := E

- Conditionals

a. "If" Statement

Two different proof rules exist depending on inclusion or exclusion of an else part.

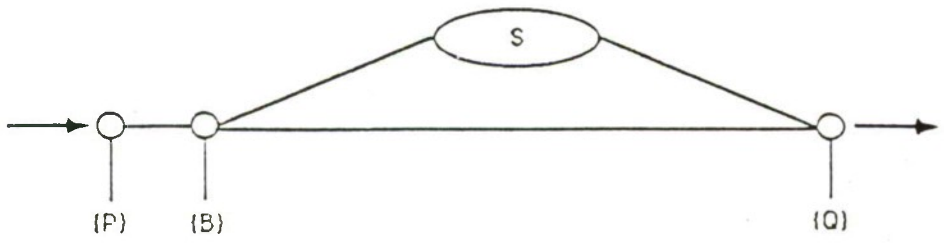
```
(({P and B} S {Q}) and ({P and ¯B} => {Q})) =>

{P}
if B then
  S
end if
{Q}
```

In order to prove the post-condition follows from the pre-condition plus the processing, two things must be proved:

1. The pre-condition P along with the condition B (equal to true in this case) and the results of processing S must imply the post-condition Q.

2. The pre-condition along with the condition B equal to false must imply the post-condition Q.

Graphic Representation:



if...then...endif

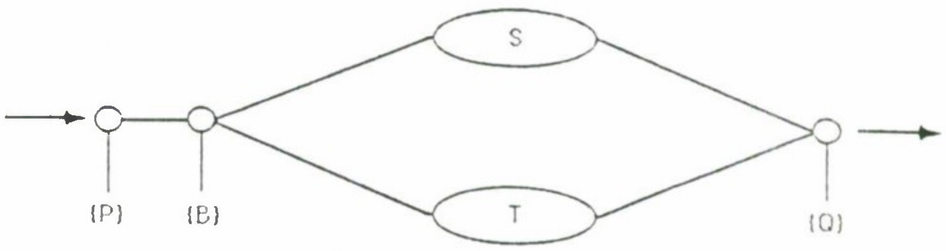
or

```
({P and B} S {Q}) and ({P and ~B} T {Q}) =>

{P}
if B then
  S
else
  T
end if
{Q}
```

1. The pre-condition P, the condition B if true, and the results of processing S must imply the post-condition Q.

2. The pre-condition P, the condition B if false, and the results of processing T must imply the post-condition Q.

Graphic Representation:



if, then, else, endif

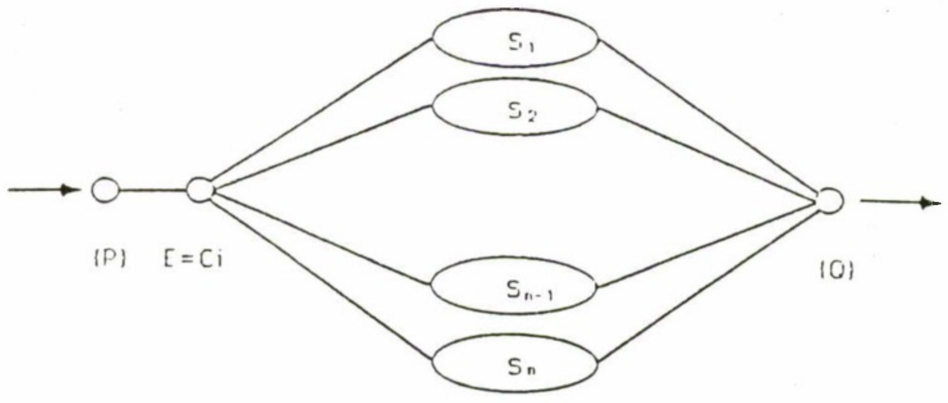
b. "Case" Statement

"Case" statements are similar to "if" statements in many ways. The notation E=Ci is used to imply that the expression E has a value which is selected by the choice Ci; this is also used to cover the 'when others' alternative if it is present.

```
for each i {P and (E = Ci)} Si {Q} =>

{P}
case E is
  when C1 => S1
  when C2 => S2
  .
  .
  end case
{Q}
```

For each of the possible values of the "case" switch E, the pre-condition together with the current value and the processing associated with the current value must imply the post-condition.

Graphic Representation:



case E is when .. end case

- "Goto" Statement

  A compound statement S containing the statement goto <<L>>, and not containing the label <<L>>.

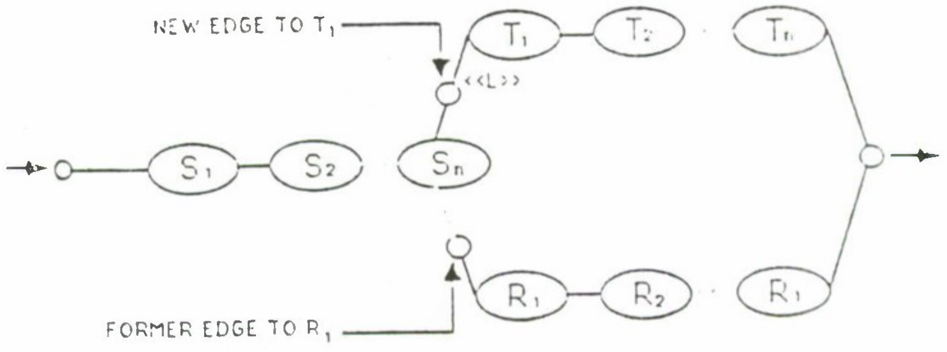
  {P} S {Q} or at <<L>> {R} =>

  If pre-condition P is true

  1. S terminates normally and {Q} is true

  2. Execution of S terminates with a 'goto L' and {R} is true

Graphic Representation:



$S_n \equiv$ goto <<L>>

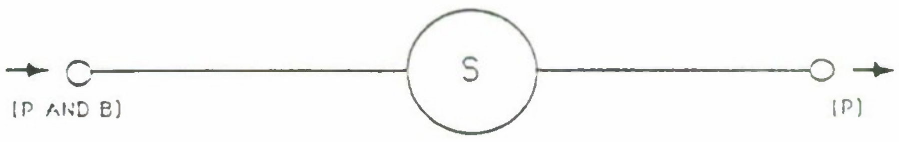
- "Loop" Statement

a. "While" Statement

```
{P and B} S {P} =>

{P}
while B
  loop
    S
  end loop
{P and ~B}
```

The pre-condition together with the processing must show that the "while" condition B is no longer true and the "loop" invariant P is still true after (possibly repeated) processing of statement S.

Graphic Representation:



while ... loop ... end loop

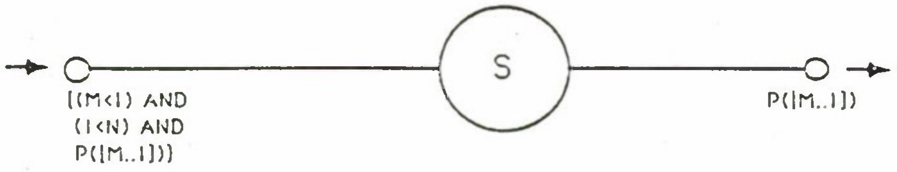
b. **"For" Statement**

The antecedent of this rule states that for all I between M and N, if the predicate P is true for all steps up to (but not including) I, and the statement S is carried out, then the predicate P will be true for I.

(standard notation '..' is used to indicate intervals)

```
    (M <= I) and (I <= N) and {P([M..I))}  S  {P([M..I])}
    =>

    {P([])}
    for I in M..N
      loop
        S
      end loop
    {P([M..N])}
```

Graphic Representation:



for ... in .. loop ... end loop

- **"Exit" Statement**

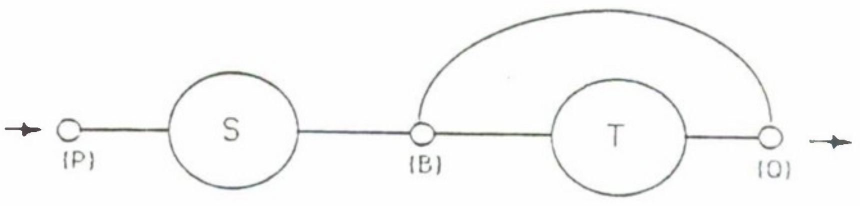
```
    ({P} S {Q}) and ({Q and B} T {P}) =>

    {P}
    loop
      S
      exit when ~B;
      T
    end loop
    {Q and ~B}
```

Two possibilities need to be proved:

1. The pre-condition P together with the processing of statement S imply the post-condition Q.

2. The post-condition Q together with the condition B not false and the processing of statement T imply the "loop" invariant P.

Graphic Representation:



loop .. exit when .. end loop

- **Concatenation**

This rule describes what is involved in combining two (or more) statements.

{P} S {Q} and {Q} T {R} =>

{P} S T {R}

If two statements are represented by

{P} S {Q}, {R} T {U}

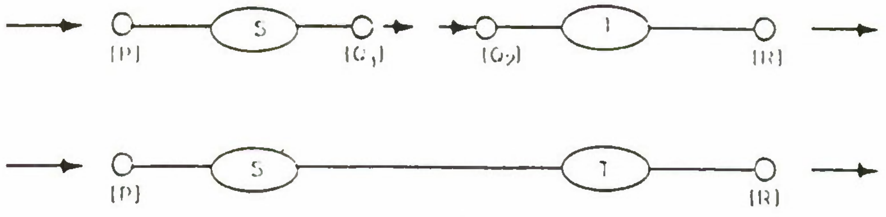
then the statements may be combined if

Q => R

Giving the statement

{P} S T {U}

Graphic Representation:



$S,T; Q_1 => Q_2$

- Declarations

a. Type Declaration

Each type declaration results in the generation of assertions to be added to the set H. The collection of declarations D is processed until all type declarati have been removed and their corresponding axioms added the set H.

H includes all the attributes and associated axioms, etc. associated with the type T. (Subtype declarations are similar)

```
H {P} D begin S end {Q} =>

{P}
  type T is ...;
  D
begin
  S
end
{Q}
```

b. Uninitialized Variable

H contains information about X derived from its declaration; this includes its attributes and the axioms the attributes satisfy. The symbol Tu stands for an undefined variable of type T.

```
H {P and X' = Tu} D(X|X') begin S(X|X') end {Q} =>

{P}
  X : T;
  D
begin
  S
end
{Q}
```

Each declaration of a variable results in the addition of assumptions available to the theorem prover. In the case of an uninitialized variable X,

1. An assertion is made that the variable X′ has the attributes of an uninitialized variable of type T

2. X is replaced throughout the block by X′

c. **Initialized Variable**

H has the same meaning as in the rule for uninitialized variables.

```
H {P and X′ = E} D(X|X′) begin S(X|X′) end {Q} =>

{P}
  X : T := E;
  D
begin
  S
end
{Q}
```

1. An assertion is made that the variable X′ has the attributes of an variable of type T, and is equal to the value of E.

2. X is replaced throughout the block S by X′.

d. **Constant**

```
{P} D(X|E) begin S(X|E) end {Q} =>

{P}
  X : constant T := E;
  D
begin
  S
end
{Q}
```

X is replaced throughout the block S by the constant value E.

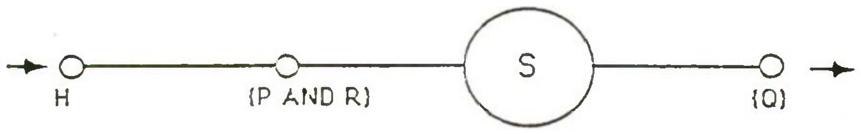
- **Blocks**

D represents a set of declarations of possibly different varieties and S represents a sequence of statements.

```
H {P and R} S {Q} =>

{P}
D
begin
  S
end
{Q}
```

H represents the information including the set of axioms and attributes derived from the declarations contained in D. R represents the set of initializations derived from the declarations of D.

Graphic Representation:



begin ... end

- **Subprograms**

a. **Function**

A pre-condition will be associated with the function that describes the range of values of and the relationships between the variables. There will also be a post-condition of some kind that describes the result obtained by calling the function.

The proof of correctness of a function definition is independent of the proof of correctness for a call to that function. When the function is defined, a proof is carried out showing that if certain pre-conditions are met, then the processing within the function will guarantee that certain post-conditions will hold. When the function is called, then the proof that the pre-conditions are met is sufficient to prove that the post-conditions will hold.

A skeleton declaration of F can be written as

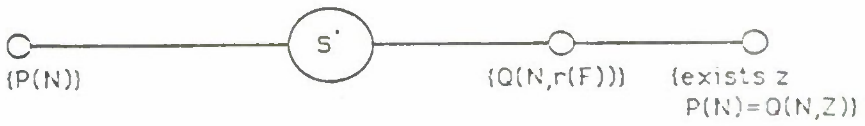
```
    function F(N: ...) return ... is
    --pre {P(N)}
    --post {Q(N,r(F))}
    ...
    begin
    S
    end F;
```

The proof rule for a function call F(A) of function F can be expressed as

```
    ({P(N)} S'; <<END>> null; {Q(N,r(F))})

    and (exists Z s.t.  P(N) => Q(N,Z))=>

    P(N)(N|A) => Q(N,r(F))(N|A,r(F)|F(A))
```

Each function definition is maintained as a distinct graph. The representative top-level graph for a function is:

Graphic Representation:



function .. return .. is .. begin .. end

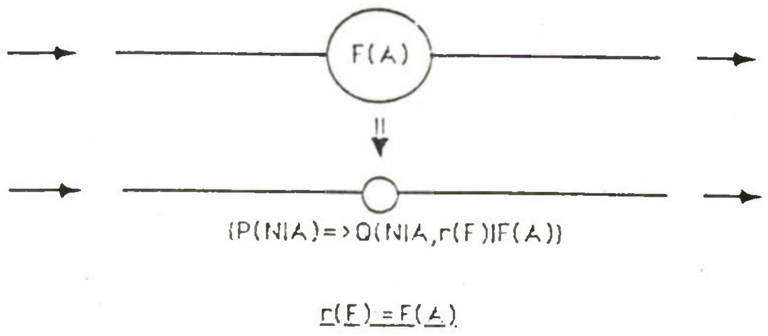
The antecedent of the rule for functions states that:

1. If the input requirements are met, then the value returned by the function will meet the output requirement.

2. For every valid input N, there is a function value that satisfies the output requirement.

The consequent of the rule for functions states that if the input conditions have been met using the actual parameters, then the post-condition will hold for those actuals. For a call of the function the graph is manipulated in the following manner:

Graphic Representation:



b. Recursive Function

To cope with recursion it is necessary to add to the antecedent a variant of the consequent, such as

```
P(N)(N|A1) =>

Q(N,r(F))(N|A1,r(F)|F(A1))
```

This formula states that if an actual A1 is supplied to the function's formal parameter N, then the output assertion will be true when the parameter is replaced by A1 and the return value of F is replaced by the recursive call of the function.

It is possible to extend this formula to functions of several variables, as well as to mutually recursive functions. In order to prove total correctness, it has to be shown that the evaluation of all parameters will terminate and that there is some ordering associated with the values of successive parameters. On successive calls, the values of the parameter must belong to a well-founded set, and must decrease.

c. Procedure

Assuming three formal parameters M, N, and O the procedure T is written

```
procedure T(M: in ...; N: in out ...; O: out ...) is
-- pre {P(M,N)}
-- post {Q(M,N,O)}
...
begin
 S
```

```
    end T;
```

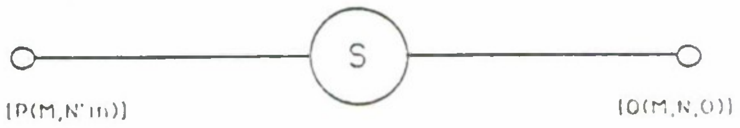
The proof rule for the procedure call T is

```
    {P(M,N'in)} S {Q(M,N'in,N'out,O)} =>

    {P(M|A,N|B,O|C)}
    T(A,B,C)
    {Q(M|A,N|B,O|C)}
```

If the pre-condition P(M,N'in) of the procedure T can be satisfied using the substitutions (M|A), (N|B), (O|C), then the post-condition will be true (using the same substitutions).

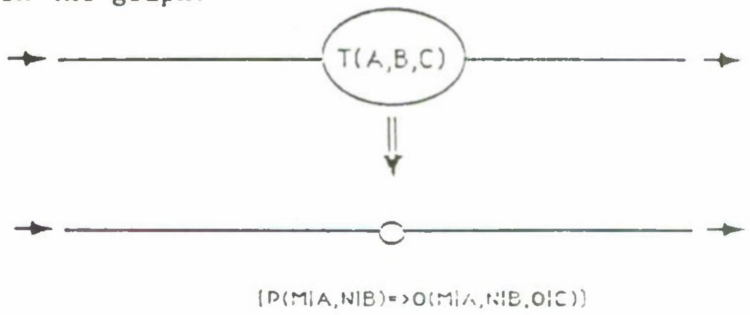
As with function definitions, each procedure definition is represented by a distinct graph:

Graphic Representation:



procedure ... is ... begin ... end

Each call of a procedure causes the following alteration in the graph:



T(A,B,C);

The parameters must be shown to satisfy the type conditions and constraints. These checks can be left to the compiler. As with functions the antecedent of the implication need be proven only once, with only the entry condition needing to be proved for each call to the procedure.

- "Return" Statement

  Each return statement

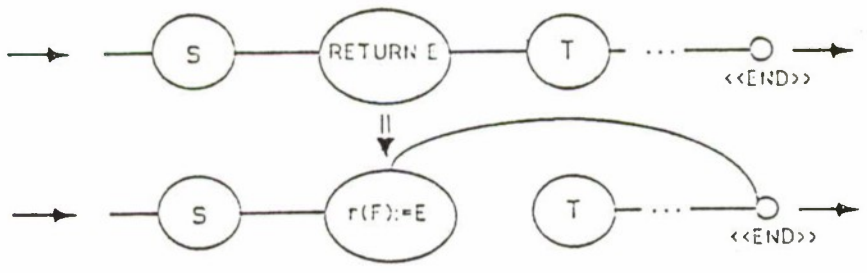
  ```
  return E;
  ```

  that appears within a function is replaced by the pair of statements

  ```
  r(F) := expression;
  goto END;
  ```

  where <<END>> (a unique label generated by STOF) and a null statement are placed at the end of the definition of the function.

  Graphic Representation:



return E

- **Globals**

  Use of globals within a subprogram can be treated in the same manner as parameters if the following rules are used:

  a. Globals accessed for reading purposes only should be treated as Ada "in" parameters.

  b. Globals that are updated directly, or indirectly using another subprogram, should be regarded as Ada 'in out' parameters.

- **Exceptions**

  The effect of a block in which an exception may occur is equivalent to the effect of either:

  a. The normal execution of the statements of the block.

b. The normal execution of the statements up to the point where the exception is raised, followed by the effect of executing the appropriate exception handler.

Case (a) does not require any modifications to the graph, because the flow of control is not modified if there is no exception raised.

Case (b) requires the following modifications of the graph.

a. Each point in the graph that could lead to the raising of an exception must be identified, along with the particular exception that would be raised.

b. The exception handlers for each possible exception must be identified within the graph.

c. The equivalent of an 'if...then...endif' is inserted at the point where the exception could occur.

   1. The node representing the conditional part of the graph holds the condition that will cause an exception.

   2. The 'then' part of the graph, which is followed if the conditions causing an exception to be raised are met, is represented by a jump to the appropriate exception handler. This is shown as an edge in the graph leading to the part of the graph holding the handler.

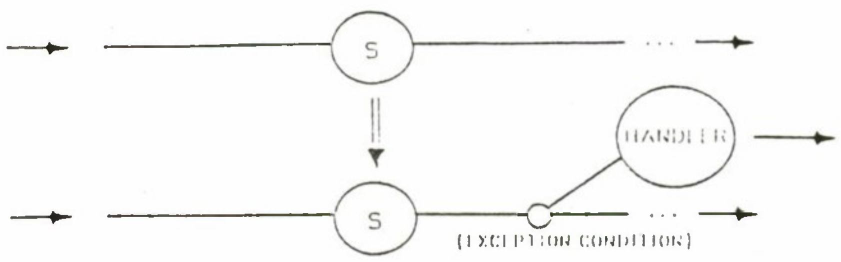
   3. If the conditions leading to an exception are not met, then the normal flow of control is followed.

The following exceptions are predefined in Ada:

CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, TASKING_ERROR

Of this list, all but TASKING_ERROR need to be considered by STOF.

Graphic Representation:



exceptions

6.2.3 STOF Ada Security Analyzer

The specification of the STOF Ada Security Analyzer is the same as that already specified for STOS since both operate on Ada source code input. (See Section 6.2.2.)

6.2.4 STOF Security Graph Generator

The Security Graph Generator takes the parse tree as input and creates a digraph for use by both the MLS Formula Generator and the Proof of Correctness Formula Generator. This graph depicts all of the components and interactions that exist in the system being analyzed. The resulting graph contains nodes that represent statements (components) and arrows that represent relationships (interactions) (see Figure 6-8).

- Description

  The use of graphical analysis of a system for MLS has been demonstrated by the program BTOS. A description of this program appears in the paper "Examination of Multi-Level Security From Data Flow Graphs," presented at the 1985 AFCEA Conference on Physical and Electronic Security, held in Philadelphia, PA. [9] An extension of the techniques used in BTOS are appropriate for use in STOF.

  The same flow graph may be used by both the MLS flow analysis and the proof of correctness analysis. The information stored in the graph is manipulated differently depending on the type of analysis being performed.

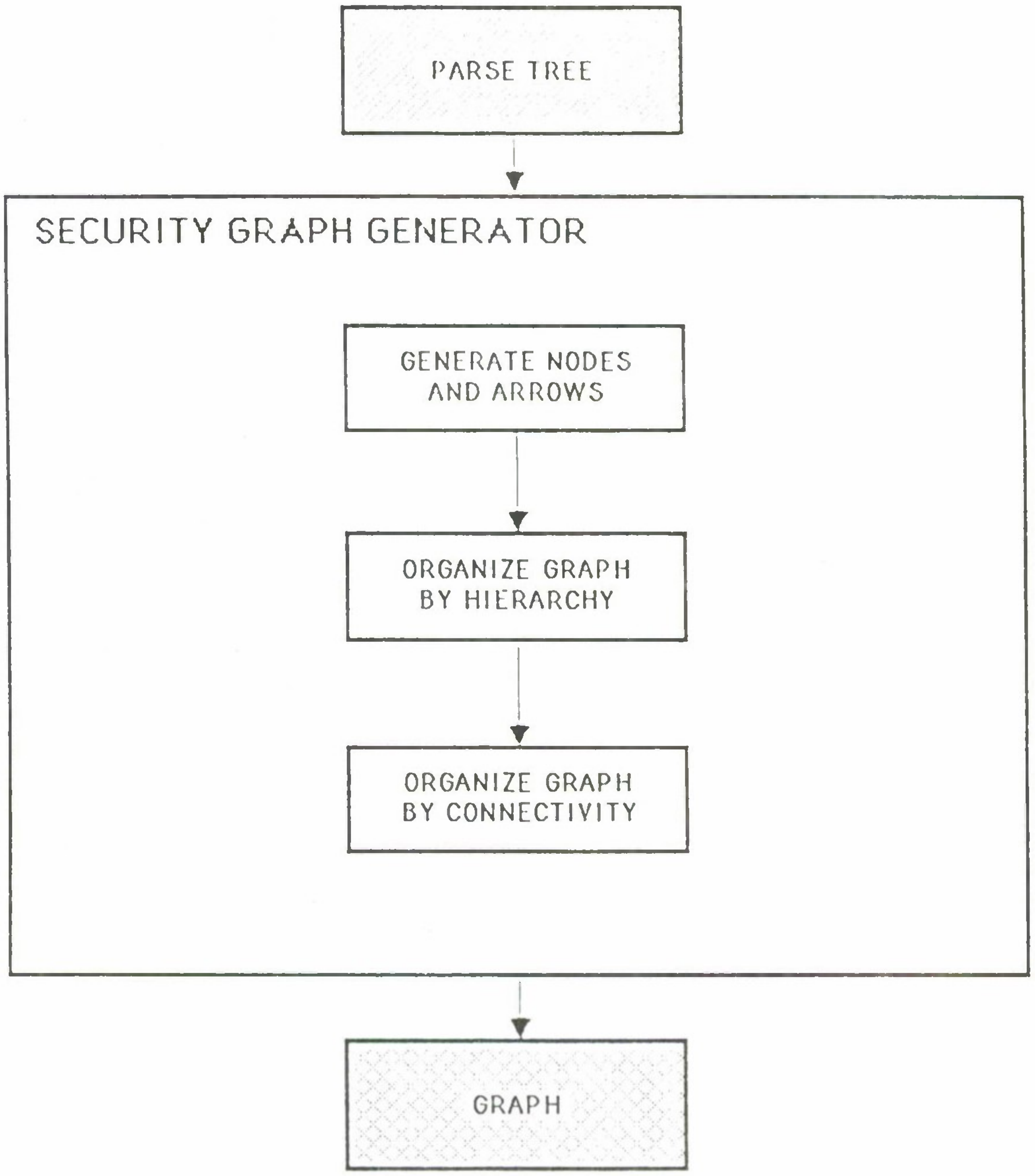

PARSE TREE
SECURITY GRAPH GENERATOR
GENERATE NODES AND ARROWS
ORGANIZE GRAPH BY HIERARCHY
ORGANIZE GRAPH BY CONNECTIVITY
GRAPH

Figure 6-8. Security Graph Generator

Nodes are generated for the following constructs (these represent the verifiable subset of the Ada language):

null, assignment, conditional, goto, loop, exit, return, block, procedure/function calls, procedure/function bodys, type/variable/constant declarations, exceptions

- o <u>Specification</u>

The graph of two successive statements is made up of the composition (concatenation) of the graphs of the two statements. Complex statements are represented by a hierarchy of nodes where the subnodes are the component statements making up the complex one. [26]

The following steps are used in transforming the annotated source into the graph.

- A node is constructed for each logical condition and for each statement. The types of nodes that can be constructed are:

  a. Declarations

     type/variable/constant
     procedure/function

  b. Pre-conditions

  c. Statements

  d. Post-conditions

  Declarations are treated separately from other statements because, rather than transforming data, they initialize values and add information to the knowledge base available when performing proofs on the associated statements.

- Compound statements are decomposed into a hierarchy of nodes. The top-most node represents the compound statement and each subnode represents an individual statement within the compound statement. Subnodes that are themselves representatives of compound statements are further decomposed in the same manner.

  Decomposition continues until the statement is atomic (cannot be further decomposed). Atomic statements in the verifiable subset are those that do not contain other statements, such as: null, assignment, goto, and exit.

- Statements that cause jumps in the flow of control (such as goto, exit, raise, return) are analyzed and the edges of the graph are modified to show the jump.

6.2.5 STOF Verification Condition Generator

- <u>MLS Flow Analyzer</u>

  The MLS Flow Analyzer is a VCG that reads the digraph output by the Security Graph Generator, and creates a set of formulas to submitted to the Formatter. The first step is to analyze all paths of the graph for information flow dependencies. [27] During this phase, labeling and propagation of the data items occurs. The existing flow dependencies along with labeling information are represented in the list of formulas generated by the MLS Flow Analyzer (see Figure 6-9).

  - <u>Description</u>

    The formal model is a statement of the security policy in a formal mathematical language. The mathematical language must be sufficiently powerful to embody the concepts presented in the security policy. The language must also be amenable to proofs showing the consistency of the model with the policy and the internal consistency of the model. Statements made in the mathematical language must be usable with little or no translation as requirements of the verification process.

  - <u>Specification</u>

    The MLS proof of secure behavior proceeds in the following manner:

    a. The source is analyzed to determine all paths through which information can flow.

    b. All information dependencies are collected as a list of pairs (X,Y), where information flows from X to Y (Y depends on X). Along with each pair there may be a set of logical statements that form a pre-condition for flow of information from X to Y.

    c. For each pair, a formula is produced in the form: P => lteq(X,Y). This is read as 'if the condition P is true then the security label of X must be less than or equal to the security label of Y'. P represents the collected

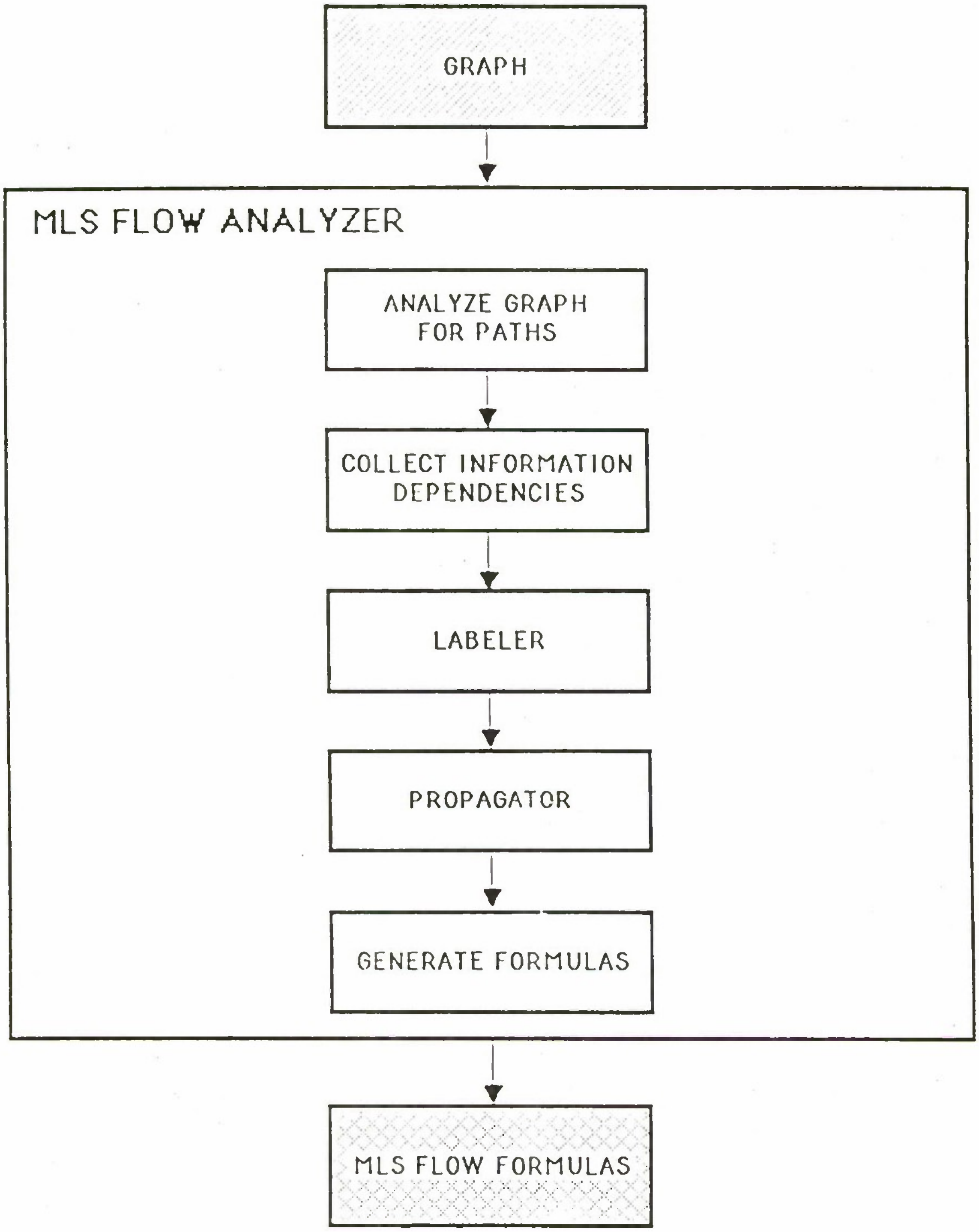


Figure 6-9. MLS Flow Analyzer Components

control/type/etc. information that the flow X -> Y depends on.

d. Each of the formulas is passed to a theorem prover, which can be fully automated, interactive, or human.

e. If all formulas generated by steps 1-3 are proved 'true' by the theorem prover, then the system is secure (according to the security model, which is implemented by steps 1,2).

- Proof of Correctness Analyzer

The Proof of Correctness Analyzer is a VCG, and performs some basic functions similar to the MLS Flow Analyzer. It reads the digraph resulting from the Security Graph Generator and creates a set of formulas to be submitted to the Formatter. The first step is to analyze the graph to find all paths. Logical conditions that appear on the paths are analyzed to create proof of correctness formulas (see Figure 6-10).

- Description

Verification conditions (VCs) for a program are composed of logical formulas from from two sources: the underlying axioms of the programming language, and the logical conditions imposed by the program specifications. The correctness of a program is the problem of solving the series of logical formulas that lead from the program entry to the program exit.

Syntactically correct programs are made up of a composition of atomic statements such as assignment and null. Each of these operations can be axiomatized and represented as nodes in a digraph. Program specifications are added to these nodes as pre-conditions, post-conditions, etc.

The process of generating VCs for a program is that of finding all paths through the representative graph of the program and collecting the logical conditions that appear on those paths.

- Specification

The entry conditions for a block of processing describe the conditions that the input must meet in order for the output to be meaningful. If the entry conditions are not met, then the use of the block cannot be considered correct. If the entry conditions are met, and the block has been previously

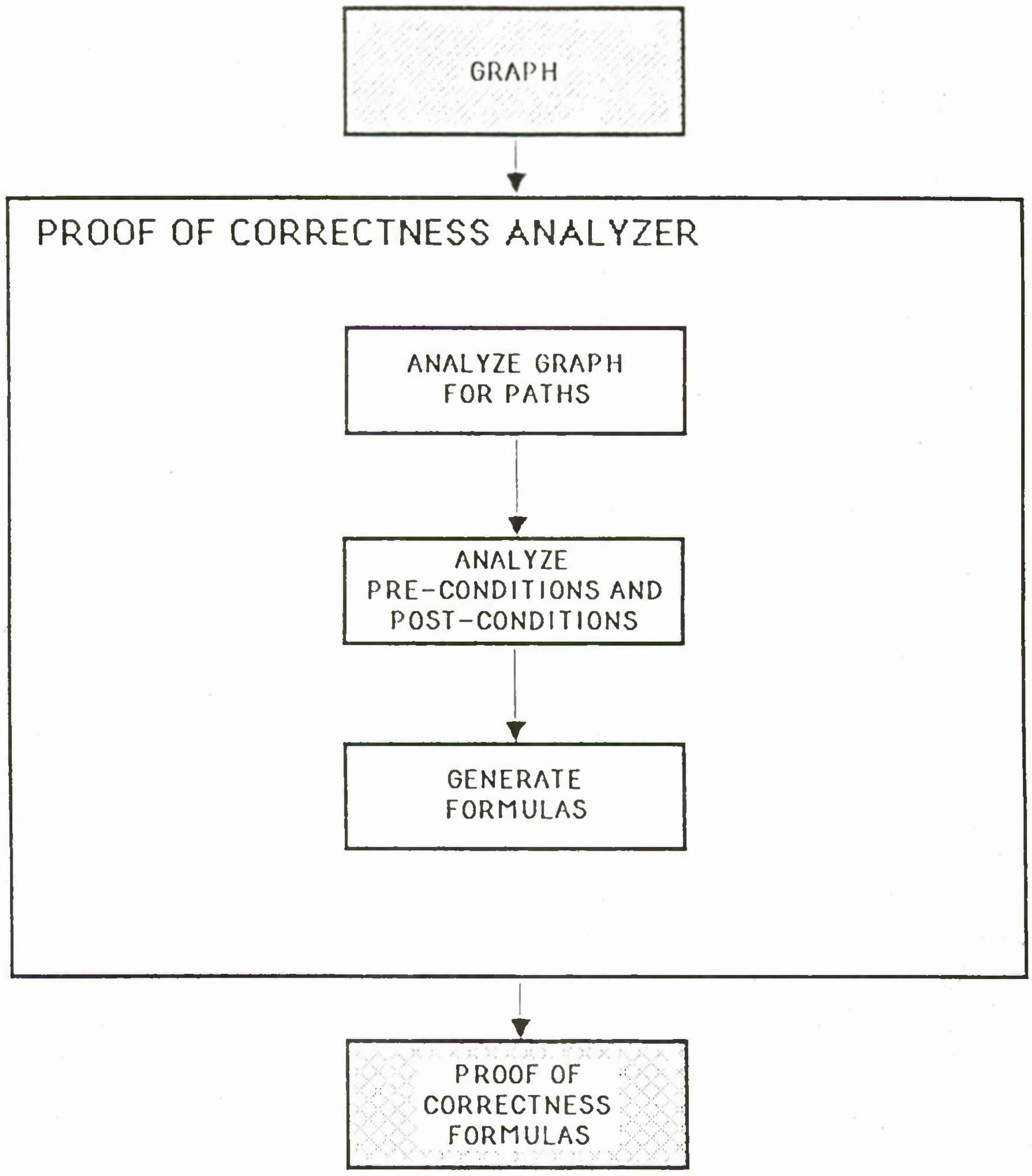

GRAPH
PROOF OF CORRECTNESS ANALYZER
ANALYZE GRAPH FOR PATHS
ANALYZE PRE-CONDITIONS AND POST-CONDITIONS
GENERATE FORMULAS
PROOF OF CORRECTNESS FORMULAS

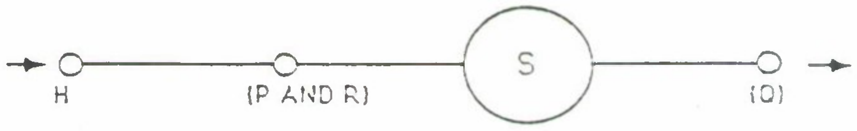
Figure 6-10. Proof of Correctness Analyzer

verified to be correct, then the output conditions of the block will hold.

The format (see below) for a proof of correctness of a single Ada block is

H {P and R} S {Q},

where H represents attribute and axiom information derived from declarations, {P} is the set of pre-conditions, {R} represents initialization values, S is the processing and {Q} is the set of post-conditions that must be satisfied. This format (or an abbreviated version) is used throughout. The graphical representation of such a block is:



begin end

The specification formally describes the net transformation of inputs into outputs. The code must be proven to produce exactly that transformation. Using the input conditions as assumptions, together with any relevant axioms about the data structures being used, the processing is analyzed to determine what inferences can be made about the data after the block is finished. If the processing is correct, then the combination of the effects of the processing and the initial assumptions and axioms should suffice to prove that the exit conditions hold.

These are conditions that need to be met after a segment of code has been analyzed and its effects have been calculated. The exit conditions describe the relationships between and on data items after processing has concluded. When connecting several blocks of code, the exit conditions of a block should imply the entry conditions of the next block. If the entry conditions plus processing imply the exit conditions in every block, and the exit conditions of each block imply the entry conditions of the following block, then the entire collection will be correct.

Some problems exist with proving total correctness of a system. [28],[10] There are examples of programs that work, but for which proofs are elusive. The following program (which is an implementation of an open problem in number theory) calculates the number of steps it takes to reduce a positive integer to 1 using the following rules:

If the number is 1 then stop.

If a number is even divide it by 2.

If a number is odd, multiply it by 3 and add 1.

This recurrence relation works (apparently) for all integers greater than zero. There is no known proof that this routine will always terminate. For all values that have been tried the routine works. The implementation of the above rules in Ada would be:

```
function step(i : integer) return integer is
begin
if (i = 1) then
   return 1;
elsif (i mod 2) = 0 then
   return step(i / 2) + 1;
else
   return step((3 * i) + 1) + 1;
end if;
end step;
```

This sort of problem would not be acceptable for analysis by mechanical theorem proving because of the nature of the parameter i. In order to show that the function terminates, it must be possible to show that there is a well-founded relation on the parameter such that the value of the relation goes down after every invocation of the function. [2]

The standard less than (<) for integers cannot be used as the well-founded relation on i, due to the possibility that a recursive call will be made using ((3 * i) + 1), which (for positive integers) is larger than i. A well-founded relation for i would have to show that each successive recursive call of the function is nearer to returning a result than the last.

The properties of a proof of correctness for this particular program are:

a. The pre-condition for this program is simple: i > 0.

b. The post-condition is also simple: the return value is exactly the depth of recursive calls of the function.

c. Proof of partial correctness is possible, as it corresponds to a proof that the depth of recursion (the number of steps taken) reached by the function is the number returned by the function.

d. Proof of termination of this algorithm is at present an open problem.

e. Therefore, a proof of total correctness cannot be produced without a proof of termination.

- <u>STOF Formatter</u>

  The Formatter organizes the formulas generated by both the MLS Flow Analyzer and the Proof of Correctness Analyzer. Output from this phase is one file containing all of the formulas generated. This file serves as input to the theorem prover (see Figure 6-11).

  - <u>Description</u>

    During this phase, any necessary modifications or enhancements to the formulas can be made. Such enhancements could include audit information showing the origin of a formula by procedure name and statement. Also, any modifications that may be necessary to accommodate a specific theorem prover can be made at this time. Thus, the porting of STOF to different theorem prover environments is easily accomplished.

  - <u>Specification</u>

    The Formatter will take as input the formulas generated by both the MLS Flow Analyzer and the Proof of Correctness Analyzer. If desired, audit information will be appended to each formula. Modifications that may be needed to accommodate a specific theorem prover will be performed. These formulas will be organized into a formulas file to be submitted to the theorem prover for analysis.

6.2.6 STOF Environment Support Tool

The Environment Support Tool interacts with the Ada Security Analyzer, the Security Graph Generator, and the VCG. Its function is to automate a thorough and accurate configuration management of all databases in the STOF environment (see Figure 6-12).

o <u>Description</u>

The Environment Support Tool controls and maintains STOF data. Several databases will need to be maintained throughout the life

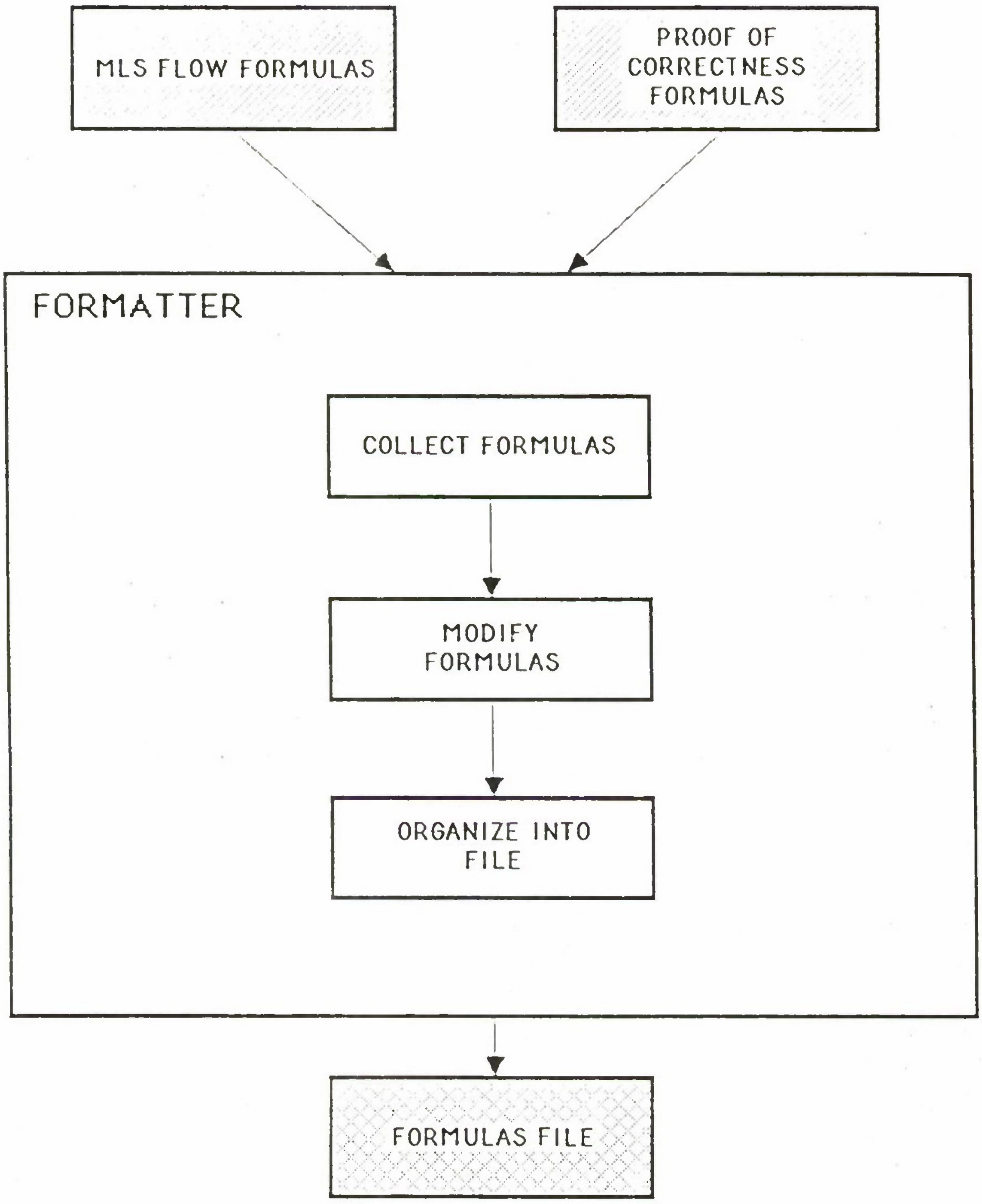


Figure 6-11. Formatter Components

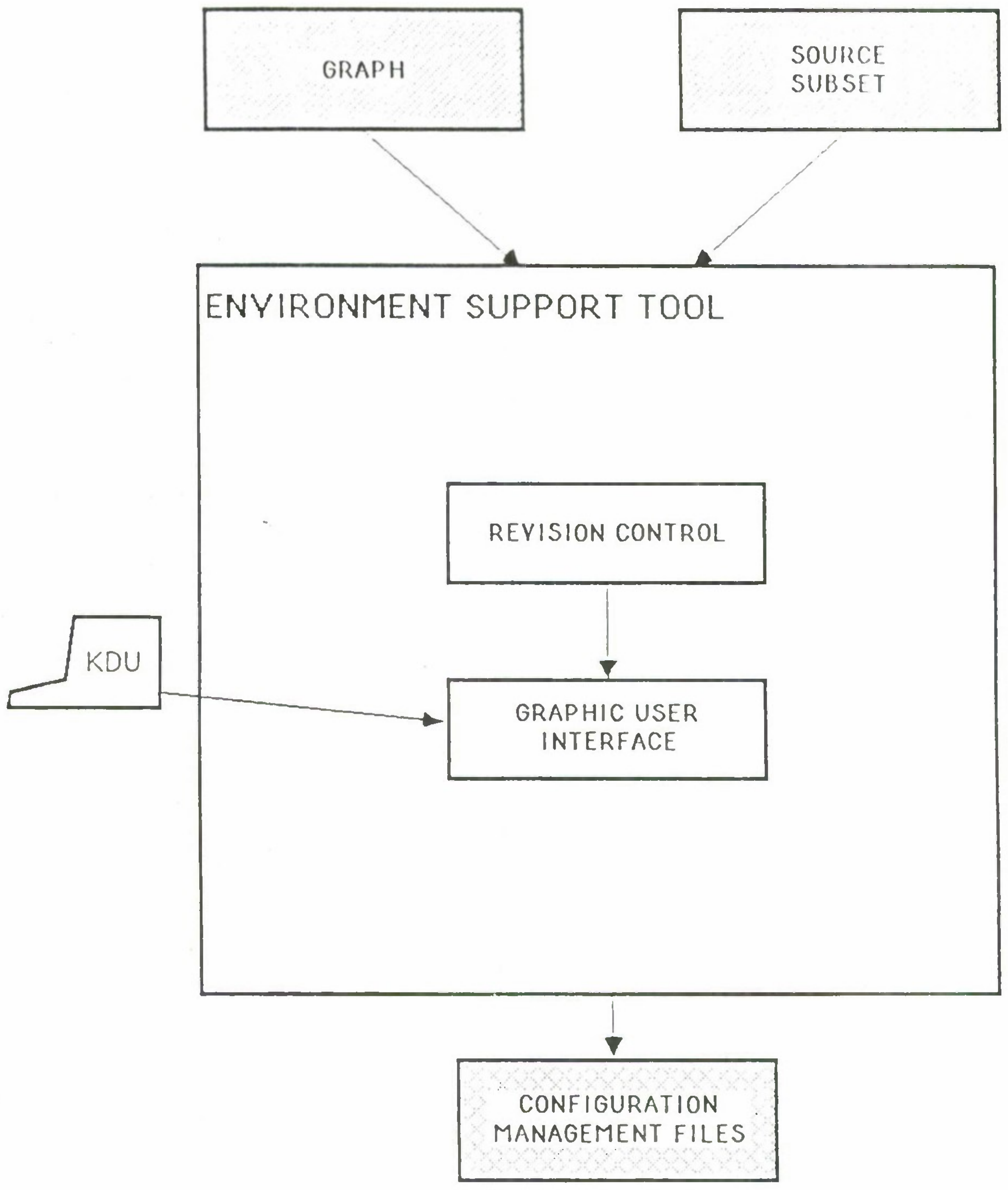


Figure 6-12. Environment Support Tool

of the STOF verification process. This tool also manages the STOF user interface. Ease of use of the STOF approach by either designers or verifiers will be determined by the power of this tool. STOF may be used by programmers who require feedback about security implications of their code, or it may be used by verification analysts or members of accreditation or certification teams. Version control and database consistency will also be addressed by this tool.

- <u>Specification</u>

  The STOF Environment Support Tool controls and maintains STOF data. This tool will be graphically oriented, and will use windows and color. The following tasks are proposed as part of the Phase II effort:

  - Design capability to perform configuration management on STOF files and data structures.

  - Design capability to maintain source code verification history.

  - Design capability for manipulating context of user view of the generated digraph.

    - Identify externally referenced modules.

    - Track dependencies.

SECTION 7

CONCLUSIONS AND RECOMMENDATIONS

7.1 CONCLUSIONS

A major advantage to the STOS translation method is the fact that the language SPECIAL is generated. SPECIAL files provide the user with an intermediate, readable form which can be analyzed and modified. Also, automated translation to SPECIAL facilitates timely use of HDM's MLS Tool and Boyer-Moore theorem prover. This accredited toolset provides an excellent means for representing information flow. Also, STOS could provide the user with less unknowns than STOF, while integrating with an existing, accredited methodology.

One major disadvantage of STOS is that the Boyer-Moore theorem prover and the HDM methodology itself have limitations. HDM is primarily an information flow analysis tool, with a built-in MLS model. As such, HDM is limited in the type of verification applications it supports. For example, HDM could be used for proof of correctness, but the amount of work this task would require would not justify the results. Another disadvantage is that the entire STOS approach takes more CPU time than the STOF approach, because the generation of a specification language is bypassed with STOF. STOS will severely limit the type of verification that can be performed on the source. Its disadvantages can create significant time and effort costs.

Results of COMPUSEC's Phase I research show that an STOF approach to verification of source code is more favorable than an STOS approach. STOF offers innovative technical progress in the field of software verification. It can be used to translate a larger subset of the Ada language than that possible using STOS. It can generate formulas for both MLS information flow and proof of correctness. It can be tailored to specific security models and theorem provers. Finally, STOF can be implemented to perform quickly with a minimum of user intervention. The Phase II effort will therefore focus on implementing an STOF system.

The following conclusions have resulted from Phase I analysis:

- STOS
  - The STOS approach can support translation of a subset of the Ada language.
  - Although the STOS approach integrates with a proven, existing verification methodology, it can be limiting. Security analysis using the HDM methodology is restricted to information flow--proof of correctness cannot be handled.
  - Questions have surfaced concerning HDM's placement on the NCSC's Endorsed Tools List (ETL). These questions will affect the recognition and acceptance of analysis accomplished using STOS.
  - Using STOS and HDM requires large computer resources and even more significant time and human intervention in order to produce proofs.

- STOF
  - STOF can be used to translate a larger subset of the Ada language than that possible using STOS.
  - STOF can generate formulas for both multilevel secure (MLS) information flow and proof-of-correctness.
  - STOF formulas can potentially be tailored to the requirements of a specific security model.
  - STOF output can be formatted to integrate with a particular theorem prover (i.e., Boyer-Moore, Shostak, or COMPUSEC).
  - STOF can be implemented on a small hardware/software configuration (i.e., a single-user workstation).
  - STOF can be implemented to perform quickly with a minimum of user intervention.
  - STOF could be submitted to the NCSC for inclusion on the ETL.
  - STOF represents genuine technical improvements for the application of verification techniques to real systems.

7.2 RECOMMENDATIONS

Results of COMPUSEC's Phase I research show that an STOF approach to verification of source code is more favorable than an STOS approach. STOF offers innovative technical progress in the field of software verification. It can be used to translate a larger subset of the Ada language than that possible using STOS. It can generate formulas for both MLS information flow and proof of correctness. It can be tailored to specific security models and theorem provers. Finally, STOF can be implemented to perform quickly with a minimum of user intervention. The Phase II effort will therefore focus on implementing an STOF system.

SECTION 8

REFERENCES

1. Feiertag, R. (1980). "A Technique for Proving Specifications are Multilevel Secure". CSL-109, SRI International, Menlo Park, CA.

2. Purdom, Paul Walton Jr., and Brown, Cynthia A. (1985). The Analysis of Algorithms. New York, NY: CBS Publishing.

3. Boyer, Robert S. and Moore, J. Strother. (1979). A Computational Logic. Academic Press, Inc., A Subsidiary of Harcourt Brace Jovanovich, Publishers, NY.

4. Carnap, Rudolf (1958). Introduction to Symbolic Logic and Its Applications. New York, NY: Dover Publications, Inc.

5. Kowalski, Robert (1979). Logic for Problem Solving. New York, NY: Elsevier Science Publishing Co., Inc.

6. Kemmerer, Richard A. (1986). Verification Assessment Study Final Report. Volumes II,IV, and V. National Computer Security Center, Office of Research and Development, Fort George G. Meade, Maryland 20755-6000.

7. CSC-EPL-85/001. (1985). "Final Evaluation Report of SCOMP" Secure Communications Processor. STOP Release 2.1. Department of Defense.

8. National Computer Security Center (1986). "Verification Support." Draft NCSC Standard Operating Procedure, October 1986, Fort Meade, MD.

9. Enzmann, Alexander R. (1985). "Examination of Multi-Level Security From Data Flow Graphs." Philadelphia, PA: AFCEA Conference on Electronic and Physical Security.

10. Hunt III, H.B. and Rosenkrantz, D.J. (August 1986). "Recursion Schemes and Recursive Programs are Exponentially Hard to Analyze." SIAM Journal on Computing. Volume 15, Number 3. Philadelphia, PA: SIAM Publications.

11. Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. (1986). Compiler Principles, Techniques, and Tools. Addison-Wesley Publishing Company, Mass.

12. ANSI/MIL-STD-1815A-1983. (1983). Reference Manual for the Ada Programming Language. United States Department of Defense. Springer-Verlag, NY.

13. Brodie, Leo. (1981). Starting FORTH. Englewood Cliffs, NJ: Prentise-Hall, Inc.

14. Clocksin, W.F., and Mellish, C.S. (1984). Programming in PROLOG. Second Edition. Berlin, Germany: Springer-Verlag.

15. Feuer, Alan and Gehani, Narain, Editors. (1984). Comparing and Assessing Programming Languages Ada, C, and Pascal. Prentice-Hall, Inc., Englewood Cliffs, NJ.

16. Kernighan, Brian W. and Ritchie, Dennis M. (1978). The C Programming Language. Prentice-Hall, Inc., Englewood Cliffs, NJ.

17. McDermid, John and Ripken, Knut. (1984). Life Cycle Support in the Ada Environment. Cambridge University Press, Cambridge, UK.

18. Winston, P.H., and Horn, B.K.P (1981). LISP. Reading, MA: Addison-Wesley Publishing Company.

19. Teitelman, Warren. (1974). Interlisp Reference Manual. Palo Alto Research Centers, 3333 Coyote Hill Road, Palo Alto, CA 94304.

20. Hume, J.N.P., and Holt, R.C. (1979). Programming FORTRAN 77: A Structured Approach. Reston, VA: Reston Publishing Company, Inc., A Prentice-Hall Company.

21. Silverberg, B., L. Robinson, and K. Levitt. (1980). The HDM Handbook, Volume II: The Languages and Tools of HDM. SRI International, Menlo Park, CA.

22. Crow, Judith, et al. (1985). SRI Verification System Version 2.0 User's Guide. California: SRI International Computer Science Laboratory.

23. Robinson, L. (1979). The HDM Handbook, Volume I: The Foundations of HDM. SRI International, Menlo Park, CA.

24. Luckham, David C. and von Henke, Friedrich W. (1984). "An Overview of Anna, a Specification Language for Ada". 1985 International Workshop in Software Specification and Design, August 1985, London, UK.

25. McGettrick, A.D. (1982). Program Verification Using Ada. New York, NY: Cambridge University Press.

26. DeMarco, Tom (1979). Structured Analysis and System Specification. New York, NY: Yourdon, Inc.

27. Hennie, Fred (1977). Introduction to Computability. Reading, MA: Addison-Wesley Publishing Co., Inc.

28. Mehlhorn, Kurt (1984). Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness. Berlin, Germany: Springer-Verlag.

APPENDIX A

CANDIDATE LANGUAGE EVALUATIONS

A.1 UNWEIGHTED CRITERIA VALUE EXPLANATIONS

Unweighted Criteria Evaluations
(1 of 4)

NAME	VAL	CRITERIA
STANDARDIZATION		
MIL-STD, ANSI, ISO, Other	1	-Implementation is popular, but no clear standard.
	2	-Published standard, but may not be rigorous
	3	-Extremely rigorous standard.
CODE STRUCTURE		
Process/Task Use	1	-Very difficult to use
	2	-Generic, non-complex mechanisms for creation
	3	-Process/task creation is supported, intertask ctrl is defined by language
Flow Control	1	-Control is difficult AND mechanisms are difficult to translate
	2	-Control is difficult OR mechanisms are difficult to translate
	3	-Extensive ctrl. structures and easy to translate
Stmt. Eval. Order	1	-Very ambiguous
	2	-Some ambiguity
	3	-Deterministic, no ambiguity
Recursion	1	-Natural part of language, so is hard to analyze security
	2	-Recursion is possible
	3	-Recursion not possible, making security analysis easier
Backtracking	1	-Natural part of language, therefore hard to perform static analysis
	2	-Parts of language use this
	3	-Not normally used, easier to perform static analysis

Unweighted Criteria Evaluations
(2 of 4)

NAME	VAL	CRITERIA
AMBIGUITY		
Scoping	1	-No scoping mechanisms, difficult to resolve names
	2	-Sometime difficult to resolve names
	3	-Name can always be resolved
Type Coercion	1	-Minimal type checking
	2	-Usually INT<=>REAL coercions but some ambiguity in info flow
	3	-Rules are unambiguous, OR coercion not allowed
Parameter Passing	1	-Total absence, OR lack of clear mechanisms
	2	-Some difficulty or ambi-guity in mechanisms
	3	-Clearly defined mechanisms, often used
VISIBILITY		
Hierarchies	1	-Flat, or 1 level
	2	-2 levels
	3	-More than 2 levels
Data Hiding	1	-Not allowed in language
	2	-Implicitly defined
	3	-Explicitly definable
Data Flow	1	-Many constructs obscure data flow
	2	-Some obscurity
	3	-Source and destination of data are always known

Unweighted Criteria Evaluations
(3 of 4)

NAME	VAL	CRITERIA
DATA REPRESENTATION		
Abstract Data Types	1	-Some types exist, but can't create new types
	2	-Hard to define/create
	3	-Rich set of techniques for creating and defining
Data Separation	1	-Self-modifying code is possible
	2	-Separation of code and data is optional
	3	-Distinct separation of code and data

Unweighted Criteria Evaluations
(4 of 4)

NAME	VAL	CRITEPIA
INTERFACES		
Operating System	0	-Language is self-standing, doesn't allow OS service access
	1	-Very limited access to OS
	2	-Limited access through some predefined functions
	3	-Easy access to extensive OS services
Application Module	1	-Multiple modules not allowed, or interface is highly ambiguous
	2	-Difficult to determine inter-faces between modules
	3	-Explicit definition of all interfaces to all components
I/O	0	-Not defined as part of language
	1	-Limited constructs AND diffi-cult to perform static analysis
	2	-Limited constructs
	3	-Rich construct set with direction-specific info
User	1	-Many discreet program deve-lopment steps, hard to read code
	2	-Language supports readable code
	3	-Comprehensive environment, tools available to support ease of program construction

A.2 UNWEIGHTED LANGUAGE EVALUATIONS

Ada		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	3
MIL-STD	-MIL-STD-1815	3
ANSI		
ISO		
Other		
CODE STRUCTURE	Total:	13
Process/Task Use	-Excellent task management Handles creation and manipulation of process/tasks, Static analysis can be used on mechanisms provided	3
Flow Control	-Rich set of control structures. Static analysis can be used on them	3
Stmt. Eval. Order	-Highly deterministic, even ambiguities are defined	3
Recursion	-Supported, but language doesn't depend on it	2
Backtracking	-Used to bind context dependent identifiers	2
AMBIGUITY	Total:	9
Scoping	-Strict rules provide excellent info for static analysis. "private" construct can reduce ambiguity "generic"construct can cause it. Some context dependent var. function, and procedure instances	3
Type Coercion	-Supported, with well defined results	3
Parameter Passing	-All modes explicitly defined. Defined ambiguities in array and record passing.	3

Ada		
CRITERIA	EVALUATION	VALUE
VISIBILITY	Total:	9
Hierarchies	-One of the most structured hierarchies for program creation	3
Data Hiding	-Rich set of constructs (e.g.'package')	3
Data Flow	-Discernible through static analysis. Easily represented between modules	3
DATA REPRESENTATION	Total:	5
Abstract Data Types	-Supported	3
Data Separation	-Can be used correct-ly or incorrectly	2
INTERFACES	Total:	10
Operating System	-Well defined and flexible	3
Application Module	-Explicitly well defined	3
I/O	-Std libraries available	2
User	-Readability allows manual or machine analysis	2

BASIC		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	1
MIL-STD		
ANSI		
ISO		
Other	-MicroSoft BASIC, many other implementations	1
CODE STRUCTURE	Total:	10
Process/Task Use	-No facility for handling	1
Flow Control	-Primitive branching	1
Stmt Eval. Order	-Often line by line interpretive, but imp-lementation dependent	2
Recursion	-Not supported	3
Backtracking	-Not part of language	3
AMBIGUITY	Total:	6
Scoping	-Not there	3
Type Coercion	-Not supported	2
Parameter Passing	-One-line functions only	1
VISIBILITY	Total:	3
Hierarchies	-Not supported (flat structure)	1
Data Hiding	-Not supported	1
Data Flow	-Insufficient structures to delineate dependencies	1

BASIC		
CRITERIA	EVALUATION	VALUE
DATA REPRESENTATION	Total:	2
Abstract Data Types	-Not supported	1
Data Separation	-Language allows modification of running code	1
INTERFACES	Total:	8
Operating System	-Limited	2
Application Module	-Overlay mechanism is highly ambiguous	1
I/O	-Limited	2
User	-Provides total environment	3

C		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	
MIL-STD		
ANSI	-In progress	1
ISO		
Other	-Kernighan & Ritchie, many extensions of this	2
CODE STRUCTURE	Total:	12
Process/Task Use	-Fork() and semaphores	2
Flow Control	-Clear except for conditional exp. eval (some data flow only visible at runtime)	2
Stmt Eval. Order	-Sequential, clearly specified	3
Recursion	-Supported, but not typical-ly used	2
Backtracking	-Not part of language	3
AMBIGUITY	Total:	7
Scoping	-Strong scoping rules, but same name can be used for different instances	2
Type Coercion	-Implicit type coercion without warning or error, but strong type check is possible	2
Parameter Passing	-Call by value is only mech. provided	3
VISIBILITY	Total:	7
Hierarchies	-All subprograms at second lexical level	2
Data Hiding	-Separate compilation, static storage class	2
Data Flow	-Suitable for static analysis	2

C		
CRITERIA	EVALUATION	VALUE
DATA REPRESENTATION	Total:	4
Abstract Data Types	-Limited: supports derived and renamed data types	2
Data Separation	-Can be used or abused	2
INTERFACES	Total:	8
Operating System	-Exceptionally good (fork and exec)	3
Application Module	-Not required, linker addr. resolution obscure	1
I/O	-Std. libraries available, but no built-in facility	2
User	-Many steps in dev. process. 'Several (many!) pass' compilers	2

FORTH		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	1
MIL-STD		
ANSI		
ISO		
Other	-FORTH Interest Group some std. functions	1
CODE STRUCTURE	Total:	11
Process/Task Use	-Stackframe maintenance or Concurrent FORTHs	2
Flow Control	-Clearly defined, except 'LEAVE' construct allows jump to end of loop structures	1
Stmt Eval. Order	-Strictly reverse polish, but can change or effect code in mid-operation	2
Recursion	-Not part of language	3
Backtracking	-Not part of language	3
AMBIGUITY	Total:	5
Scoping	-All vars and subprograms are global. Difficult to resolve name or symbol	1
Type Coercion	-Not there	3
Parameter Passing	-None, except global stack(s)	1

FORTH		
CRITERIA	EVALUATION	VALUE
VISIBILITY	Total:	5
Hierarchies	-Unrestricted nesting	3
Data Hiding	-All components always accessible	1
Data Flow	-Non-deterministic for static analysis	1
DATA REPRESENTATION	Total:	3
Abstract Data Types	-Extremely difficult to implement/simulate	1
Data Separation	-Can be used well or abused	2
INTERFACES	Total:	4
Operating System	-Is small kernel, not used from OS	0
Application Module	-One program at a time, interfaces through stack only	1
I/O	-Limited and diffi- cult to analyze	1
User	-Interpretive code can be made readable	2

FORTRAN		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	2
MIL-STD		
ANSI	-FORTRAN 77	1
ISO		
Other	-WATFOR, WATFIV, FORTRANIV, et al	1
CODE STRUCTURE	Total:	11
Process/Task Use	-Very difficult using only FORTRAN constructs	1
Flow Control	-Some richness in control structures	2
Stmt Eval. Order	-Line by line, some ambiguities	2
Recursion	-Not part of language	3
Backtracking	-Not part of language	3
AMBIGUITY	Total:	6
Scoping	-Global vars. (COMMON block) Explicit local var. declaration or implicit local declaration using default type. Difficult to translate, complex static analysis	2
Type Coercion	-Explicit type change: Int.<=>Real, Left side of assgnmnt dominates mode	2
Parameter Passing	-Strictly by reference-copy in copy out	2

FORTRAN		
CRITERIA	EVALUATION	VALUE
VISIBILITY	Total:	7
Hierarchies	-Program can be 2-level tree	2
Data Hiding	-Implicit in some parts of language	2
Data Flow	-No indirect flows in assignment stmts, COMMON block variable value relates to param. pass mech., and order of alias assgnmnt in subprogram. Mechanized translation candidate	3
DATA REPRESENTATION	Total:	3
Abstract Data Types	-Creation not supported, only existing types	1
Data Separation	-Normally separated	2
INTERFACES	Total:	5
Operating System	-Very few std. interfaces	1
Application Module	-Some checks occur only at run-time. Difficulties for static analysis	2
I/O	-Limited. Difficulties for static analysis	1
User	-Many dev. steps. Flat code not very readable	1

LISP		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	1
MIL-STD		
ANSI		
ISO		
Other	-Common LISP, Interlisp, MACLISP,Franzlisp, contain some common features	1
CODE STRUCTURE	Total:	9
Process/Task Use	-Not a natural part of the language.	1
Flow Control	-Function evaluation, loops, branching, iterated statements (i.e. for loops) are supported in some LISPs.	2
Stmt Eval. Order	-Normal evaluation is through list processing. Expressions are in unambiguous prefix notation.	3
Recursion	-Directly supported	1
Backtracking	-Certain operations implement backtracking, in particular pattern matching procedures.	2
AMBIGUITY	Total:	6
Scoping	-Variables are visible in the scope in which they are defined, and within any functions descendent from the scope in which they are defined.	2
Type Coercion	-No type checking, coercion is normally only used in arithmetic expressions.	1
Parameter Passing	-Call by value, always hands a value in	3

Lisp		
CRITERIA	EVALUATION	VALUE
VISIBILITY	Total:	10
Hierarchies	-Unrestricted nesting of function calls.	3
Data Hiding	-Some explicit functions are provided to implement it.	3
Data Flow	-Can be readily determined except for cases performing assignments through indirection (i.e. (SET A B) sets the variable pointed to by A to the value pointed to by B).	2
DATA REPRESENTATION	Total:	4
Abstract Data Types	-Support for records,arrays and hash tables is typical	3
Data Separation	-Self-modifying code is natural part of language	1
INTERFACES	Total:	9
Operating System	-Implementation dependent, normally little interaction happens with the OS other than file manipulation.	1
Application Module	-No checks except at run-time	2
I/O	-Extensive IO and text manipulation functions are provided.	3
User	-Debugging and editing functions are part of the language.	3

MODULA-2		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	3
MIL-STD		
ANSI		
ISO		
Other	-Nicklaus Wirth	3
CODE STRUCTURE	Total:	12
Process/Task Use	-Loosely coupled processes Ideal for data flow analy- sis	3
Flow Control	-All the mechs. supported by PASCAL, plus a few more Use of ptrs can obscure flow	2
Stmt Eval. Order	-Sequential, not rigorously defined	2
Recursion	-Supported	2
Backtracking	-Not part of language	3
AMBIGUITY	Total:	9
Scoping	-Well defined, suitable for static analysis	3
Type Coercion	-Coercion is definable	3
Parameter Passing	-Well defined, call by ref, call by value	3
VISIBILITY	Total:	9
Hierarchies	-Unrestricted; foundation of language	3
Data Hiding	-Explicitly supported	3
Data Flow	-Excellent mechs. for data flow analysis	3

MODULA-2		
CRITERIA	EVALUATION	VALUE
DATA REPRESENTATION	Total:	5
Abstract Data Types	-Rich constructs	3
Data Separation	-Can be used or abused	2
INTERFACES	Total:	8
Operating System	-No well defined method of access to OS; can be used to create self-standing sys	0
Application Module	-All interfaces are explicit and suited for static analysis	3
I/O	-Uses standard utility modules	3
User	-Highly interactive, supports checking, yields readable code	2

PASCAL		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	2
MIL-STD		
ANSI	-Standard exists	1
ISO	-Standard exists	1
Other		
CODE STRUCTURE	Total:	11
Process/Task Use	-Only available in extended versions	2
Flow Control	-Well defined, but ptrs. can obscure flow	2
Stmt Eval. Order	-Sequential, not well defined, but most implementations follow standard parsing	2
Recursion	-Supported	2
Backtracking	-Not part of language	3
AMBIGUITY	Total:	8
Scoping	-Well defined, suitable for data flow analysis	3
Type Coercion	-Some support	2
Parameter Passing	-Well defined, call by ref, call by value	3

PASCAL		
CRITERIA	EVALUATION	VALUE
VISIBILITY	Total:	8
Hierarchies	-More than 2 levels of subprograms possible	3
Data Hiding	-Supported in simple and derived types, also by using scoping	2
Data Flow	-Suitable for static analysis	3
DATA REPRESENTATION	Total:	5
Abstract Data Types	-Supported	3
Data Separation	-Can be used or abused	2
INTERFACES	Total:	8
Operating System	-No exception handling facility specified	2
Application Module	-Modular compilation not supported in std., but most implementations support it	2
I/O	-Not part of original specification	2
User	-Supports a lot of checks, yields readable code	2

PL/M		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	3
MIL-STD		
ANSI		
ISO		
Other	-Intel language	3
CODE STRUCTURE	Total:	13
Process/Task Use	-Process/task management is supported to some extent	2
Flow Control	-PASCAL+C combination, rich constructs, easy to translate	3
Stmt. Eval. Order	-Somewhat ambiguous	2
Recursion	-Supported, but can be turned off	3
Backtracking	-Not part of language	3
AMBIGUITY	Total:	7
Scoping	-Well defined rules will aid data flow analysis	3
Type Coercion	-Pointer types cause ambiguity	1
Parameter Passing	-Call by value, mech. well-defined	3
VISIBILITY	Total:	8
Hierarchies	-More than 2 levels of subprograms possible	3
Data Hiding	-Supported at the subprogram level, like PASCAL	2
Data Flow	-Suitable for static analysis	3

PL/M		
CRITERIA	EVALUATION	VALUE
DATA REPRESENTATION	Total:	3
Abstract Data Types	-Limited. C-like	2
Data Separation	-Doesn't support data/code separation	1
INTERFACES	Total:	9
Operating System	-Can call predefined OS routines, or install user defined exception handlers	3
Application Module	-Well defined, deterministic	3
I/O	-Not language relevant	0
User	-Customizable for individual applications, yields readable code	3

PROLOG		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	1
MIL-STD		
ANSI		
ISO		
Other	-Clocksin and Mellish, but non-standard predicates abound, many different environments	1
CODE STRUCTURE	Total:	10
Process/Task Use	-Program is a conjunction of goals satisfiable through use of processes	3
Flow Control	-Fact causes immediate satisfaction of goal, reduces task to subgoals	2
Stmt. Eval. Order	-Attempts left to right 3 attribs: position, precedence class, and associativity. Brackets or associativity disambiguate expressions	3
Recursion	-Recursive goal instantiations automatically evaluated	1
Backtracking	-Initiated when goal cannot be satisfied	1
AMBIGUITY	Total:	6
Scoping	-Non-deterministic, can make translation difficult, variable value accessible only within single clause	2
Type Coercion	-Type checking not usual, Arith. assgnmnt must be numbers for rule to succeed	1
Parameter Passing	-Well-defined mechanism used to pass values between program parts	3

PROLOG		
CRITERIA	EVALUATION	VALUE
VISIBILITY	Total:	5
Hierarchies	-All definition at 1 level	1
Data Hiding	-Implicit in parts of language	2
Data Flow	-Several clauses may affect same data	2
DATA REPRESENTATION	Total:	3
Abstract Data Types	-Limited	2
Data Separation	-Code and data are same	1
INTERFACES	Total:	5
Operating System	-Self-standing, doesn't interface with OS	0
Application Module	-Doesn't deal with program modules, rather with sets of data	1
I/O	-Limited	1
User	-Provides comprehensive user environment	3

SNOBOL		
CRITERIA	EVALUATION	VALUE
STANDARDIZATION	Total:	1
MIL-STD		
ANSI		
ISO		
Other	-Many different implementations	1
CODE STRUCTURE	Total:	8
Process/Task Use	-No mechanisms	1
Flow Control	-Branching depends on success or failure of pattern matching, structured code not possible	1
Stmt. Eval. Order	-Line by line, conditionally on success of rule Left to right within order of precedence, except exponentiation (right to left)	2
Recursion	-Not possible	3
Backtracking	-Not explicit, but used for pattern matching of strings. Some ctrl possible using 'FAIL','FENCE',and'ABORT'	1
AMBIGUITY	Total:	8
Scoping	-Name can refer to other names	2
Type Coercion	-Not applicable	3
Parameter Passing	-Strictly call by value	3

SNOBOL		
CRITERIA	EVALUATION	VALUE
VISIBILITY	Total:	6
Hierarchies	-2 levels of functions exist	2
Data Hiding	-Implicit	2
Data Flow	-Indirection is often used; hard to follow	2
DATA REPRESENTATION	Total:	3
Abstract Data Types	-Non-existent	1
Data Separation	-Can be used or abused	2
INTERFACES	Total:	4
Operating System	-Limited, difficult to implement	1
Application Module	-Can be defined, but not usually part of lang.	1
I/O	-Limited to RW of strings	1
User	-Many steps in dev. process	1

A.3 WEIGHTED LANGUAGE EVALUATIONS

CRITERIA	ADA	BASIC	C	FORTH	FORTRAN	LISP	MODULA	PASCAL	PLM	PROLOG	SNOBOL
STANDARDIZATION (10)	15	1	2	1	3	1	3	4	3	1	1
MIL-STD	3	0	0	0	0	0	0	0	0	0	0
x5	15	0	0	0	0	0	0	0	0	0	0
ANSI	0	0	0	0	1	0	0	1	0	0	0
x2	0	0	0	0	2	0	0	2	0	0	0
ISO	0	0	0	0	0	0	0	1	0	0	0
x2	0	0	0	0	0	0	0	2	0	0	0
Other	0	1	2	1	1	1	3	0	3	1	1
x1	0	1	2	1	1	1	3	0	3	1	1
CODE STRUCTURE (20)	57	28	43	48	33	30	51	41	48	49	26
Process/Task Use	3	1	2	2	1	1	3	2	2	3	1
x10	30	10	20	20	10	10	30	20	20	30	10
Flow Control	3	1	2	1	2	2	2	2	3	2	1
x5	15	5	10	5	10	10	10	10	15	10	5
Stmt. Eval. Order	3	2	3	2	2	3	2	2	2	3	2
x2	6	4	6	4	4	6	4	4	4	6	4
Recursion	2	3	2	3	3	1	2	2	3	1	3
x2	4	6	4	6	6	2	4	4	6	2	6
Backtracking	2	3	3	3	3	2	3	3	3	1	1
x1	2	3	3	3	3	2	3	3	3	1	1

CRITERIA	ADA	BASIC	C	FORTH	FORTRAN	LISP	MODULA	PASCAL	PLM	PROLOG	SNOBOL
AMBIGUITY (20)	60	46	50	28	40	46	60	56	52	46	54
Scoping x6	3 18	3 18	2 12	1 6	2 12	2 12	3 18	3 18	3 18	2 12	2 12
Type Coercion x4	3 12	2 8	2 8	3 12	2 8	1 4	3 12	2 8	1 4	1 4	3 12
Parameter Passing x10	3 30	1 10	3 30	1 10	2 20	3 30	3 30	3 30	3 30	3 30	3 30
VISIBILITY (15)	45	15	36	25	36	39	45	41	41	25	30
Hierarchies x5	3 15	1 5	2 10	3 15	2 10	3 15	3 15	3 15	3 15	1 5	2 10
Data Hiding x4	3 12	1 4	2 8	1 4	2 8	3 12	3 12	2 8	2 8	2 8	2 8
Data Flow x6	3 18	1 6	3 18	1 6	3 18	2 12	3 18	3 18	3 18	2 12	2 12
DATA REPRESENTA-TION (15)	40	15	30	35	20	35	40	40	25	25	20
Abstract Data Types x10	3 30	1 10	2 20	3 30	1 10	3 30	3 30	3 30	2 20	2 20	1 10
Data Separation x5	2 10	1 5	2 10	1 5	2 10	1 5	2 10	2 10	1 5	1 5	2 10

CRITERIA	ADA	BASIC	C	FORTH	FORTRAN	LISP	MODULA	PASCAL	PLM	PROLOG	SNOBOL
INTERFACES (20)	53	34	37	17	30	42	43	40	45	19	20
Operating System x5	3 15	2 10	3 15	0 0	1 5	1 5	0 0	2 10	3 15	0 0	1 5
Application Module x8	3 24	1 8	1 8	1 8	2 16	2 16	3 24	2 16	3 24	1 8	1 8
I/O x5	2 10	2 10	2 10	1 5	1 5	3 15	3 15	2 10	0 0	1 5	1 5
User x2	2 4	3 6	2 4	2 4	2 4	3 6	2 4	2 4	3 6	3 6	1 2

A.4 SUMMARY LANGUAGE EVALUATION

CRITERIA	ADA	MODULA	PASCAL	PLM	C	LISP	PROLOG	FORTRAN	FORTH	SNOBOL	BASIC
TOTALS	270	242	222	214	198	193	165	163	154	151	139
STANDARDIZATION (10)	15	3	4	3	2	1	1	3	1	1	1
CODE STRUCTURE (20)	57	51	41	48	43	30	49	33	48	26	28
AMBIGUITY (20)	60	60	56	52	50	46	46	40	28	54	46
VISIBILITY (15)	45	45	41	41	36	39	25	36	25	30	15
DATA REPRESENTA-TION (15)	40	40	40	25	30	35	25	20	35	20	15
INTERFACES (20)	53	43	40	45	37	42	19	30	17	20	34

APPENDIX B

SAMPLE STOF EXAMPLE

B.1 ADA SOURCE CODE

```
-- Function to compute the greatest common divisor of a,b.

function gcd(a,b : natural) return natural is
   -- pre(true)
   -- post(r(gcd) = gcd(a,b))

   u : integer := a;
   v : integer := b;
   w : integer;
begin
   while (v > 0) loop
      -- {(gcd(a,b) = gcd(u,v))}
      w := u mod v;
      u := v;
      v := w;
   end loop;

   return u;
end gcd;
```

B.2 GRAPH OF GCD GENERATED BY STOF

(See Figure B-1)

B.3 PROOF OF CORRECTNESS OF GCD

There are two parts to the proof of correctness: proof of partial correctness, and proof of termination. The formulas for the proof of correctness are taken directly from the conditions and assumptions appearing on the representative graph of the function. These formulas, when proven, show that the preconditions of gcd plus the processing in gcd imply the postconditions of gcd.

I) FORMULAS

The following formulas must be proven true in order to show the partial correctness of gcd:

A) The first formula generated establishes that if the entry conditions for the procedure along with information deduced from the type and variable declarations are correct, then the preconditions for processing are met.

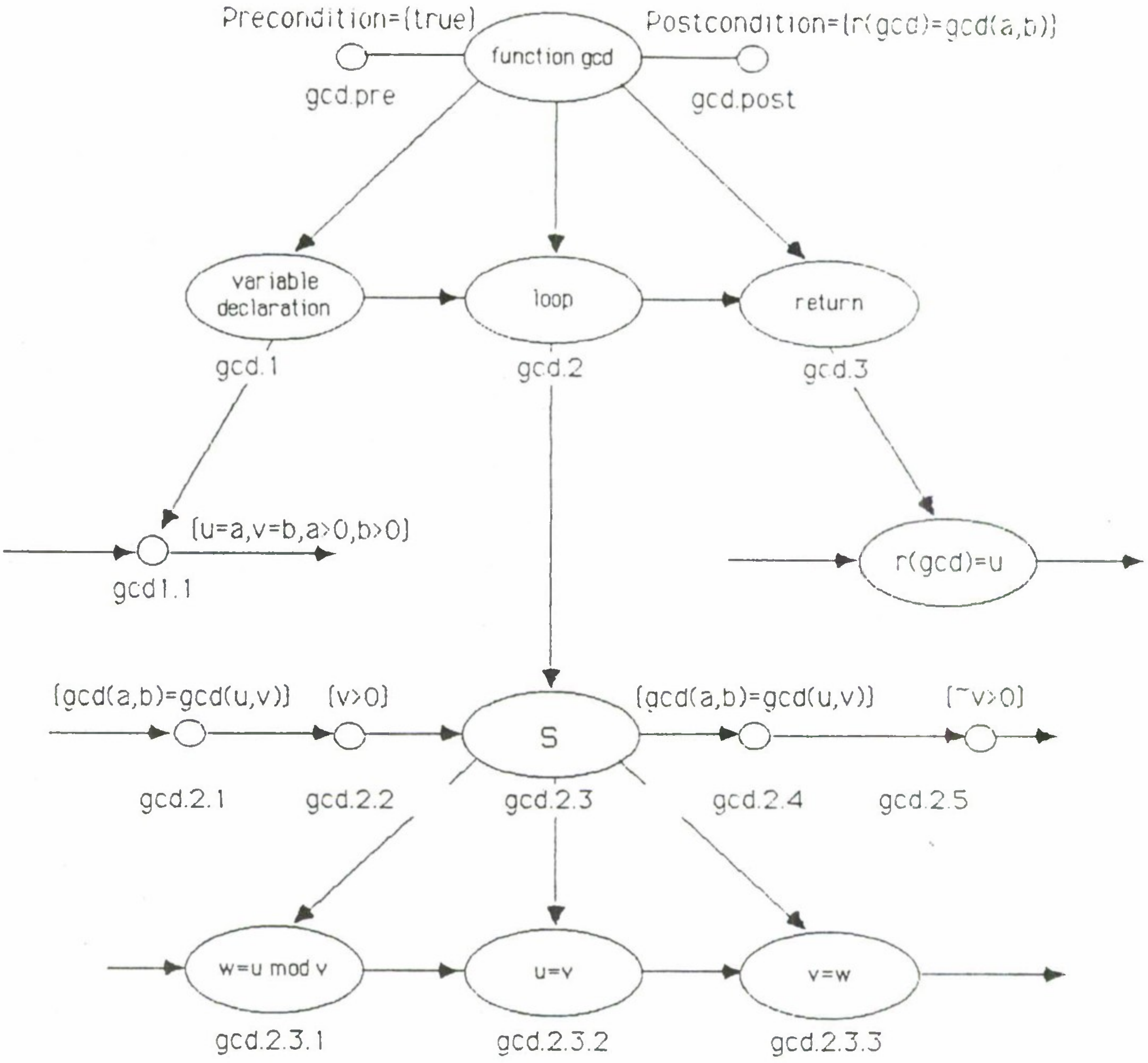


Figure B-1. STOF Graph of GCD

```
{true and (u = a) and (v = b) and (a > 0) and (b > 0)} =>
{gcd(a,b) = gcd(u,v)}
```

The lhs of the formula is generated from the precondition for the function gdc, and from the variable and type decalarations node (gcd.1.1). The rhs of the formula is from the preconditions for the first statement in the procedure (gcd.2.1 and gcd.2.2).

B) The second formula establishes that if the preconditions for the loop statement are met, and the loop condition is true, then the processing of the loop will maintain the loop invariant.

```
{(gcd(a,b) = gcd(u,v)) and (v > 0)}
[(w = u mod v)(u = v)(v = w)] => {(gcd(a,b) = gcd(u,v)}
```

The lhs of the formula is generated from the preconditions for the loop (gcd.2.1) and the loop ocndition (gcd.2.2). The rhs is generated from the postcondition for the loop (gcd.2.4) which is also referred to as the loop invariant.

C) The third formula establishes that when the loop is terminated the loop invariant is true (which shows that the processing within the loop is correct) and the postconditions for the loop statement are true.

```
{(gcd(a,b) = gcd(u,v)) and ~(v > 0)} =>
{(gcd(a,b) = gcd(u,v)) and (v = 0)}
```

The lhs is generated from the loop invariant (gdc.2.1) and the negation of the loop condition (gcd.2.2). The rhs is from the loop invariant (gcd.2.4) and the termination condition for the loop (gcd.2.5).

D) The fourth formula establishes that if the precondtions for the return statement are met, and the expression in the return statement is evaluated then the postconditions for the procedure gcd will be met.

```
{(gcd(a,b) = gcd(u,v)) and (v = 0)}[r(gcd) = u] =>
{r(gcd) = gcd(a,b)}
```

The lhs is generated from the postconditions for theloop statement (gcd.2.4 and gcd.2.5) and the processing in the return statement (gcd.3.1). The rhs is generated from the postcondition for the procedure gcd.

II) PROOFS

A) The type information about the parameters a and b was used to deduce the following:

$a > 0,\ b > 0$

The variable declarations added the following:

$u = a,\ v = b$

Substitution of a for u, and b for v shows

$\gcd(a,b) = \gcd(u,v)$

B) In order to prove the loop invariant, (gcd(a,b) = gcd(u,v)), it must be demonstrated that it remains valid after each cycle of the loop.

The series of steps through the loop produce the following transformations to the variables u,v,w:

i) $w = u \bmod v,$
$u = v,$
$v = w \Rightarrow v = u \bmod v$

By substitution, the formula becomes:

$\{\gcd(a,b) = \gcd(u,v))$ and $(v > 0)\} \Rightarrow$
$\{\gcd(a,b) = \gcd(v, u \bmod v)\}$

The following facts about gcd are true:

a) $\gcd(x,0) = \gcd(0,x) = x$

b) $\gcd(m,n) = \gcd(n,m) = \gcd(n,m - n)$

By induction,

ii) $(m - q*n >= 0) \Rightarrow (\gcd(n,m) = \gcd(n,m - q*n))$

The function mod is defined as follows:

If $m,n > 0$ then (by rules of algebra)

iii) $m = n*q + rem$, where $0 <= rem < n$, and
$m \bmod n = rem$.

Given values for q, rem which satisfy iii,

$u \bmod v = rem$, and $rem >= 0$,

which satisfies the second part of the loop invariant.

Using the substitutions in i, and the results in ii and iii,

```
gcd(v,u mod v) = gcd(v,rem)
               = gcd(v,v - q*u)
               = gcd(v,u)
               = gcd(u,v)
               = gcd(a,b).
```

C) In order to show:

```
{(gcd(a,b) = gcd(u,v)) and ~(v > 0)} =>
{(gcd(a,b) = gcd(u,v)) and (v = 0)}
```

It suffices to show

```
~(v > 0) => (v = 0)
```

Since after each cycle of the loop,

```
v = u mod v,
```

By iii, (v >= 0), and since, ~(v > 0) => (v <= 0),

```
(v >= 0) and (v <= 0) => (v = 0).
```

D) At the termination of the loop, the variable v = 0, and the return value of the function is u. The post condition {r(gcd) = gcd(a,b)}, can be proved from the following substitutions of values:

```
gcd(a,b) = gcd(u,v)
         = gcd(u,0)
         = u
         = r(gcd).
```

Thus the return value of the function is the greatest common divisor of u and v, which proves the postcondition for the function gcd. Therefore the function gcd is partially correct.

III) In order to demonstrate the termination of gcd, it suffices to show the termination of the loop. The loop will terminate if (v <= 0). Notice that after each cycle,

```
v' = u mod v,
```

which by iii implies

$$v' < v$$

where v′ is the value of v after one cycle of the loop. Since b is finite and initially v = b, it will take at most b steps before v = 0. Therefore the function will terminate.

IV) Since gcd is partially correct, and terminates, it is totally correct.

GLOSSARY

ATOS	Ada-to-SPECIAL
BTOS	Bubble-to-SPECIAL
COS/NFE	Computer Operating System/Network Front-End
ETL	Endorsed Tools List
E-HDM	Enhanced HDM
FDM	Formal Development Methodology
FTLS	Formal Top-Level Specification
HDM	Hierarchical Development Methodology
ITP	Interactive Theorem Prover
JSS	Job Stream Separator
KVM	Kernalized IBM VM
lhs	Left-hand side
MLS	Multi-level Secure
NCSC	National Computer Security Center
OS	Operating System
PDL	Program Design Language
rhs	Right-hand side
STPE	Secure Transacting Processing Experiment
STOS	Source-to-SPECIAL
STOF	Source-to-Formula
TCB	Trusted Computing Base
VCG	Verification Condition Generator